Final Report for
Contract N00039-82-C-0235
ARPA Order 4031
Productivity Engineering in the UNIX† Environment

December 1985

ABSTRACT

This report summarizes the results obtained during the contract period. The first two sections of the report and the three appendices summarize work on the Berkeley Vax UNIX system. The directions for the new work for the next contract period are summarized. A major portion of this part of the report gives a description of the new file system and networking facilities that were implemented to meet the needs of the ARPA research community. The third and fourth sections summarize research in human/machine interaction and in expert database systems. This work was initiated later in the contract period.

The first section describes the basic kernel functions provided to a UNIX process: process naming and protection, memory management, software interrupts, object references (descriptors), time and statistics functions, and resource controls. These facilities, as well as facilities for bootstrap, shutdown and process accounting, are provided solely by the kernel.

The second section describes the standard system abstractions for files and file systems, communication, terminal handling, and process control and debugging. These facilities are implemented by the operating system or by network server processes. The first of three appendixes summarizes the system primitives.

The second appendix describes a reimplementation of the UNIX file system. The reimplementation provides substantially higher throughput rates by using more flexible allocation policies, that allow better locality of reference and that can be adapted to a wide range of peripheral and processor characteristics. The new file system clusters data that is sequentially accessed and provides two block sizes to allow fast access for large files while not wasting large amounts of space for small files. File access rates of up to ten times faster than the traditional UNIX file system are experienced. Long needed enhancements to the user interface are discussed. These include a mechanism to lock files, extensions of the name space across file systems, the ability to use arbitrary length file names, and provisions for efficient administrative control of resource usage.

The last appendix gives a detailed description of the internal structure of the networking facilities. These facilities are based on several central abstractions that structure the external (user) view of network communication as well as the internal (system) implementation.

The third section of the report gives a brief summary of the research initiated in February 1984 under Supplement A to the contract. Most of those projects in the areas of computer graphics, intelligent computer systems, and

† UNIX is a trademark of Bell Laboratories.

software development environments are being continued during the next contract period. The fourth section summarizes research on expert database systems initiated under Supplement B to the contract and continued during the next contract period.

TABLE OF CONTENTS

I. Unix System Enhancements

0. Notation and types

The notation used to describe system calls is a variant of a C language call, consisting of a prototype call followed by declaration of parameters and results. An additional keyword **result**, not part of the normal C language, is used to indicate which of the declared entities receive results. As an example, consider the *read* call, as described in section 2.1:

```
cc = read(fd, buf, nbytes);
result int cc; int fd; result char *buf; int nbytes;
```

The first line shows how the *read* routine is called, with three parameters. As shown on the second line *cc* is an integer and *read* also returns information in the parameter *buf*.

Description of all error conditions arising from each system call is not provided here; they appear in the programmer's manual. In particular, when accessed from the C language, many calls return a characteristic –1 value when an error occurs, returning the error code in the global variable *errno*. Other languages may present errors in different ways.

A number of system standard types are defined in the include file <sys/types.h> and used in the specifications here and in many C programs. These include **caddr_t** giving a memory address (typically as a character pointer), **off_t** giving a file offset (typically as a long integer), and a set of unsigned types **u_char**, **u_short**, **u_int** and **u_long**, shorthand names for **unsigned char**, **unsigned short**, etc.

1. Kernel primitives

The facilities available to a UNIX user process are logically divided into two parts: kernel facilities directly implemented by UNIX code running in the operating system, and system facilities implemented either by the system, or in cooperation with a *server process*. These kernel facilities are described in this section 1.

The facilities implemented in the kernel are those which define the *UNIX virtual machine* which each process runs in. Like many real machines, this virtual machine has memory management hardware, an interrupt facility, timers and counters. The UNIX virtual machine also allows access to files and other objects through a set of *descriptors*. Each descriptor resembles a device controller, and supports a set of operations. Like devices on real machines, some of which are internal to the machine and some of which are external, parts of the descriptor machinery are built-in to the operating system, while other parts are often implemented in server processes on other machines. The facilities provided through the descriptor machinery are described in section 2.

1.1. Processes and protection

1.1.1. Host and process identifiers

Each UNIX host has associated with it a 32-bit host id, and a host name of up to 255 characters. These are set (by a privileged user) and returned by the calls:

```
sethostid(hostid)
long hostid;

hostid = gethostid();
result long hostid;

sethostname(name, namelen)
char *name; int namelen;

len = gethostname(buf, buflen)
result int len; result char *buf; int buflen;
```

On each host runs a set of *processes*. Each process is largely independent of other processes, having its own protection domain, address space, timers, and an independent set of references to system or user implemented objects.

Each process in a host is named by an integer called the *process id*. This number is in the range 1-30000 and is returned by the *getpid* routine:

```
pid = getpid();
result int pid;
```

On each UNIX host this identifier is guaranteed to be unique; in a multi-host environment, the (hostid, process id) pairs are guaranteed unique.

1.1.2. Process creation and termination

A new process is created by making a logical duplicate of an existing process:

```
pid = fork();
result int pid;
```

The *fork* call returns twice, once in the parent process, where *pid* is the process identifier of the child, and once in the child process where *pid* is 0. The parent-child relationship induces a hierarchical structure on the set of processes in the system.

A process may terminate by executing an *exit* call:

```
exit(status)
int status;
```

returning 8 bits of exit status to its parent.

When a child process exits or terminates abnormally, the parent process receives information about any event which caused termination of the child process. A second call provides a non-blocking interface and may also be used to retrieve information about resources consumed by the process during its lifetime.

```
#include <sys/wait.h>

pid = wait(astatus);
result int pid; result union wait *astatus;

pid = wait3(astatus, options, arusage);
result int pid; result union waitstatus *astatus;
int options; result struct rusage *arusage;
```

A process can overlay itself with the memory image of another process, passing the newly created process a set of parameters, using the call:

```
execve(name, argv, envp)
char *name, **argv, **envp;
```

The specified *name* must be a file which is in a format recognized by the system, either a binary executable file or a file which causes the execution of a specified interpreter program to process its contents.

1.1.3. User and group ids

Each process in the system has associated with it two user-id's: a *real user id* and a *effective user id*, both non-negative 16 bit integers. Each process has an *real accounting group id* and an *effective accounting group id* and a set of *access group id's*. The group id's are non-negative 16 bit integers. Each process may be in several different access groups, with the maximum concurrent number of access groups a system compilation parameter, the constant NGROUPS in the file <sys/param.h>, guaranteed to be at least 8.

The real and effective user ids associated with a process are returned by:

```
ruid = getuid();
result int ruid;

euid = geteuid();
result int euid;
```

the real and effective accounting group ids by:

```
rgid = getgid();
result int rgid;

egid = getegid();
result int egid;
```

and the access group id set is returned by a *getgroups* call:

```
ngroups = getgroups(gidsetsize, gidset);
result int ngroups; int gidsetsize; result int gidset[gidsetsize];
```

The user and group id's are assigned at login time using the *setreuid*, *setregid*, and *setgroups* calls:

```
setreuid(ruid, euid);
int ruid, euid;

setregid(rgid, egid);
int rgid, egid;

setgroups(gidsetsize, gidset)
int gidsetsize; int gidset[gidsetsize];
```

The *setreuid* call sets both the real and effective user-id's, while the *setregid* call sets both the real and effective accounting group id's. Unless the caller is the super-user, *ruid* must be equal to either the current real or effective user-id, and *rgid* equal to either the current real or effective accounting group id. The *setgroups* call is restricted to the super-user.

1.1.4. Process groups

Each process in the system is also normally associated with a *process group*. The group of processes in a process group is sometimes referred to as a *job* and manipulated by high-level system software (such as the shell). The current process group of a process is returned by the *getpgrp* call:

```
pgrp = getpgrp(pid);
result int pgrp; int pid;
```

When a process is in a specific process group it may receive software interrupts affecting the group, causing the group to suspend or resume execution or to be interrupted or terminated. In particular, a system terminal has a process group and only processes which are in the process group of the terminal may read from the terminal, allowing arbitration of terminals among several different jobs.

The process group associated with a process may be changed by the *setpgrp* call:

```
setpgrp(pid, pgrp);
int pid, pgrp;
```

Newly created processes are assigned process id's distinct from all processes and process groups, and the same process group as their parent. A normal (unprivileged) process may set its process group equal to its process id. A privileged process may set the process group of any process to any value.

1.2. Memory management†

1.2.1. Text, data and stack

Each process begins execution with three logical areas of memory called text, data and stack. The text area is read-only and shared, while the data and stack areas are private to the process. Both the data and stack areas may be extended and contracted on program request. The call

```
addr = sbrk(incr);
result caddr_t addr; int incr;
```

changes the size of the data area by *incr* bytes and returns the new end of the data area, while

```
addr = sstk(incr);
result caddr_t addr; int incr;
```

changes the size of the stack area. The stack area is also automatically extended as needed. On the VAX the text and data areas are adjacent in the P0 region, while the stack section is in the P1 region, and grows downward.

1.2.2 Mapping pages

The system supports sharing of data between processes by allowing pages to be mapped into memory. These mapped pages may be *shared* with other processes or *private* to the process. Protection and sharing options are defined in <mman.h> as:

† This section represents the interface planned for later releases of the system. Of the calls described in this section, only *sbrk* and *getpagesize* are included in 4.2BSD.

```
/* protections are chosen from these bits, or-ed together */
#define  PROT_READ        0x4    /* pages can be read */
#define  PROT_WRITE       0x2    /* pages can be written */
#define  PROT_EXEC        0x1    /* pages can be executed */

/* sharing types; choose either SHARED or PRIVATE */
#define  MAP_SHARED       1      /* share changes */
#define  MAP_PRIVATE      2      /* changes are private */
```

The cpu-dependent size of a page is returned by the *getpagesize* system call:

```
pagesize = getpagesize();
result int pagesize;
```

The call:

```
mmap(addr, len, prot, share, fd, pos);
caddr_t addr; int len, prot, share, fd; off_t pos;
```

causes the pages starting at *addr* and continuing for *len* bytes to be mapped from the object represented by descriptor *fd*, at absolute position *pos*. The parameter *share* specifies whether modifications made to this mapped copy of the page, are to be kept *private*, or are to be *shared* with other references. The parameter *prot* specifies the accessibility of the mapped pages. The *addr*, *len*, and *pos* parameters must all be multiples of the pagesize.

A process can move pages within its own memory by using the *mremap* call:

```
mremap(addr, len, prot, share, fromaddr);
caddr_t addr; int len, prot, share; caddr_t fromaddr;
```

This call maps the pages starting at *fromaddr* to the address specified by *addr*.

A mapping can be removed by the call

```
munmap(addr, len);
caddr_t addr; int len;
```

This causes further references to these pages to refer to private pages initialized to zero.

1.2.3. Page protection control

A process can control the protection of pages using the call

```
mprotect(addr, len, prot);
caddr_t addr; int len, prot;
```

This call changes the specified pages to have protection *prot*.

1.2.4. Giving and getting advice

A process that has knowledge of its memory behavior may use the *madvise* call:

```
madvise(addr, len, behav);
caddr_t addr; int len, behav;
```

Behav describes expected behavior, as given in <mman.h>:

```
#define  MADV_NORMAL      0      /* no further special treatment */
#define  MADV_RANDOM      1      /* expect random page references */
#define  MADV_SEQUENTIAL  2      /* expect sequential references */
#define  MADV_WILLNEED    3      /* will need these pages */
#define  MADV_DONTNEED    4      /* don't need these pages */
```

Finally, a process may obtain information about whether pages are core resident by using the call

```
mincore(addr, len, vec)
caddr_t addr; int len; result char *vec;
```

Here the current core residency of the pages is returned in the character array *vec*, with a value of 1 meaning that the page is in-core.

1.3. Signals

1.3.1. Overview

The system defines a set of *signals* that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify the *handler* to which a signal is delivered, or specify that the signal is to be *blocked* or *ignored*. A process may also specify that a *default* action is to be taken when signals occur.

Some signals will cause a process to exit when they are not caught. This may be accompanied by creation of a *core* image file, containing the current memory image of the process for use in post-mortem debugging. A process may choose to have signals delivered on a special stack, so that sophisticated software stack manipulations are possible.

All signals have the same *priority*. If multiple signals are pending simultaneously, the order in which they are delivered to a process is implementation specific. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. Mechanisms are provided whereby critical sections of code may protect themselves against the occurrence of specified signals.

1.3.2. Signal types

The signals defined by the system fall into one of five classes: hardware conditions, software conditions, input/output notification, process control, or resource control. The set of signals is defined in the file <signal.h>.

Hardware signals are derived from exceptional conditions which may occur during execution. Such signals include SIGFPE representing floating point and other arithmetic exceptions, SIGILL for illegal instruction execution, SIGSEGV for addresses outside the currently assigned area of memory, and SIGBUS for accesses that violate memory protection constraints. Other, more cpu-specific hardware signals exist, such as those for the various customer-reserved instructions on the VAX (SIGIOT, SIGEMT, and SIGTRAP).

Software signals reflect interrupts generated by user request: SIGINT for the normal interrupt signal; SIGQUIT for the more powerful *quit* signal, that normally causes a core image to be generated; SIGHUP and SIGTERM that cause graceful process termination, either because a user has "hung up", or by user or program request; and SIGKILL, a more powerful termination signal which a process cannot catch or ignore. Other software signals (SIGALRM, SIGVTALRM, SIGPROF) indicate the expiration of interval timers.

A process can request notification via a SIGIO signal when input or output is possible on a descriptor, or when a *non-blocking* operation completes. A process may request to receive a SIGURG signal when an urgent condition arises.

A process may be *stopped* by a signal sent to it or the members of its process group. The SIGSTOP signal is a powerful stop signal, because it cannot be caught. Other stop signals SIGTSTP, SIGTTIN, and SIGTTOU are used when a user request, input request, or output request respectively is the reason the process is being stopped. A SIGCONT signal is sent to a process when it is continued from a stopped state. Processes may receive notification with a SIGCHLD signal when a child process changes state, either by stopping or by terminating.

Exceeding resource limits may cause signals to be generated. SIGXCPU occurs when a process nears its CPU time limit and SIGXFSZ warns that the limit on file size creation has been reached.

1.3.3. Signal handlers

A process has a handler associated with each signal that controls the way the signal is delivered. The call

```
#include <signal.h>

struct sigvec {
        int     (*sv_handler)();
        int     sv_mask;
        int     sv_onstack;
};

sigvec(signo, sv, osv)
int signo; struct sigvec *sv; result struct sigvec *osv;
```

assigns interrupt handler address *sv_handler* to signal *signo*. Each handler address specifies either an interrupt routine for the signal, that the signal is to be ignored, or that a default action (usually process termination) is to occur if the signal occurs. The constants SIG_IGN and SIG_DEF used as values for *sv_handler* cause ignoring or defaulting of a condition. The *sv_mask* and *sv_onstack* values specify the signal mask to be used when the handler is invoked and whether the handler should operate on the normal run-time stack or a special signal stack (see below). If *osv* is non-zero, the previous signal vector is returned.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it will be delivered. The process of signal delivery adds the signal to be delivered and those signals specified in the associated signal handler's *sv_mask* to a set of those *masked* for the process, saves the current process context, and places the process in the context of the signal handling routine. The call is arranged so that if the signal handling routine exits normally the signal mask will be restored and the process will resume execution in the original context. If the process wishes to resume in a different context, then it must arrange to restore the signal mask itself.

The mask of *blocked* signals is independent of handlers for signals. It prevents signals from being delivered much as a raised hardware interrupt priority level prevents hardware interrupts. Preventing an interrupt from occurring by changing the handler is analogous to disabling a device from further interrupts.

The signal handling routine *sv_handler* is called by a C call of the form

```
(*sv_handler)(signo, code, scp);
int signo; long code; struct sigcontext *scp;
```

The *signo* gives the number of the signal that occurred, and the *code*, a word of information supplied by the hardware. The *scp* parameter is a pointer to a machine-dependent structure containing the information for restoring the context before the signal.

1.3.4. Sending signals

A process can send a signal to another process or group of processes with the calls:

```
kill(pid, signo)
int pid, signo;

killpgrp(pgrp, signo)
int pgrp, signo;
```

Unless the process sending the signal is privileged, it and the process receiving the signal must have the same effective user id.

Signals are also sent implicitly from a terminal device to the process group associated with the terminal when certain input characters are typed.

1.3.5. Protecting critical sections

To block a section of code against one or more signals, a *sigblock* call may be used to add a set of signals to the existing mask, returning the old mask:

```
oldmask = sigblock(mask);
result long oldmask; long mask;
```

The old mask can then be restored later with *sigsetmask*,

```
oldmask = sigsetmask(mask);
result long oldmask; long mask;
```

The *sigblock* call can be used to read the current mask by specifying an empty *mask*.

It is possible to check conditions with some signals blocked, and then to pause waiting for a signal and restoring the mask, by using:

```
sigpause(mask);
long mask;
```

1.3.6. Signal stacks

Applications that maintain complex or fixed size stacks can use the call

```
struct sigstack {
        caddr_t     ss_sp;
        int         ss_onstack;
};

sigstack(ss, oss)
struct sigstack *ss; result struct sigstack *oss;
```

to provide the system with a stack based at *ss_sp* for delivery of signals. The value *ss_onstack* indicates whether the process is currently on the signal stack, a notion maintained in software by the system.

When a signal is to be delivered, the system checks whether the process is on a signal stack. If not, then the process is switched to the signal stack for delivery, with the return from the signal arranged to restore the previous stack.

If the process wishes to take a non-local exit from the signal routine, or run code from the signal stack that uses a different stack, a *sigstack* call should be used to reset the signal stack.

1.4. Timers

1.4.1. Real time

The system's notion of the current Greenwich time and the current time zone is set and returned by the call by the calls:

```
#include <sys/time.h>

settimeofday(tvp, tzp);
struct timeval *tp;
struct timezone *tzp;

gettimeofday(tp, tzp);
result struct timeval *tp;
result struct timezone *tzp;
```

where the structures are defined in <sys/time.h> as:

```
struct timeval {
        long    tv_sec;         /* seconds since Jan 1, 1970 */
        long    tv_usec;        /* and microseconds */
};

struct timezone {
        int     tz_minuteswest; /* of Greenwich */
        int     tz_dsttime;     /* type of dst correction to apply */
};
```

Earlier versions of UNIX contained only a 1-second resolution version of this call, which remains as a library routine:

```
time(tvsec)
result long *tvsec;
```

returning only the tv_sec field from the *gettimeofday* call.

1.4.2. Interval time

The system provides each process with three interval timers, defined in <sys/time.h>:

```
#define ITIMER_REAL     0       /* real time intervals */
#define ITIMER_VIRTUAL  1       /* virtual time intervals */
#define ITIMER_PROF     2       /* user and system virtual time */
```

The ITIMER_REAL timer decrements in real time. It could be used by a library routine to maintain a wakeup service queue. A SIGALRM signal is delivered when this timer expires.

The ITIMER_VIRTUAL timer decrements in process virtual time. It runs only when the process is executing. A SIGVTALRM signal is delivered when it expires.

The ITIMER_PROF timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by processes to statistically profile their execution. A SIGPROF signal is delivered when it expires.

A timer value is defined by the *itimerval* structure:

```
struct itimerval {
        struct  timeval it_interval; /* timer interval */
        struct  timeval it_value;    /* current value */
};
```

and a timer is set or read by the call:

```
getitimer(which, value);
int which; result struct itimerval *value;

setitimer(which, value, ovalue);
int which; struct itimerval *value; result struct itimerval *ovalue;
```

The third argument to *setitimer* specifies an optional structure to receive the previous contents of the interval timer. A timer can be disabled by specifying a timer value of 0.

The system rounds argument timer intervals to be not less than the resolution of its clock. This clock resolution can be determined by loading a very small value into a timer and reading the timer back to see what value resulted.

The *alarm* system call of earlier versions of UNIX is provided as a library routine using the ITIMER_REAL timer. The process profiling facilities of earlier versions of UNIX remain because it is not always possible to guarantee the automatic restart of system calls after receipt of a signal.

```
profil(buf, bufsize, offset, scale);
result char *buf; int bufsize, offset, scale;
```

1.5. Descriptors

1.5.1. The reference table

Each process has access to resources through *descriptors*. Each descriptor is a handle allowing the process to reference objects such as files, devices and communications links.

Rather than allowing processes direct access to descriptors, the system introduces a level of indirection, so that descriptors may be shared between processes. Each process has a *descriptor reference table*, containing pointers to the actual descriptors. The descriptors themselves thus have multiple references, and are reference counted by the system.

Each process has a fixed size descriptor reference table, where the size is returned by the *getdtablesize* call:

```
nds = getdtablesize();
result int nds;
```

and guaranteed to be at least 20. The entries in the descriptor reference table are referred to by small integers; for example if there are 20 slots they are numbered 0 to 19.

1.5.2. Descriptor properties

Each descriptor has a logical set of properties maintained by the system and defined by its *type*. Each type supports a set of operations; some operations, such as reading and writing, are common to several abstractions, while others are unique. The generic operations applying to many of these types are described in section 2.1. Naming contexts, files and directories are described in section 2.2. Section 2.3 describes communications domains and sockets. Terminals and (structured and unstructured) devices are described in section 2.4.

1.5.3. Managing descriptor references

A duplicate of a descriptor reference may be made by doing

```
new = dup(old);
result int new; int old;
```

returning a copy of descriptor reference *old* indistinguishable from the original. The *new* chosen

by the system will be the smallest unused descriptor reference slot. A copy of a descriptor reference may be made in a specific slot by doing

```
dup2(old, new);
int old, new;
```

The *dup2* call causes the system to deallocate the descriptor reference current occupying slot *new*, if any, replacing it with a reference to the same descriptor as old. This deallocation is also performed by:

```
close(old);
int old;
```

1.5.4. Multiplexing requests

The system provides a standard way to do synchronous and asynchronous multiplexing of operations.

Synchronous multiplexing is performed by using the *select* call:

```
nds = select(nd, in, out, except, tvp);
result int nds; int nd; result *in, *out, *except;
struct timeval *tvp;
```

The *select* call examines the descriptors specified by the sets *in*, *out* and *except*, replacing the specified bit masks by the subsets that select for input, output, and exceptional conditions respectively (*nd* indicates the size, in bytes, of the bit masks). If any descriptors meet the following criteria, then the number of such descriptors is returned in *nds* and the bit masks are updated.

- A descriptor selects for input if an input oriented operation such as *read* or *receive* is possible, or if a connection request may be accepted (see section 2.3.1.4).
- A descriptor selects for output if an output oriented operation such as *write* or *send* is possible, or if an operation that was "in progress", such as connection establishment, has completed (see section 2.1.3).
- A descriptor selects for an exceptional condition if a condition that would cause a SIGURG signal to be generated exists (see section 1.3.2).

If none of the specified conditions is true, the operation blocks for at most the amount of time specified by *tvp*, or waits for one of the conditions to arise if *tvp* is given as 0.

Options affecting i/o on a descriptor may be read and set by the call:

```
dopt = fcntl(d, cmd, arg)
result int dopt; int d, cmd, arg;

/* interesting values for cmd */
#define  F_SETFL        3       /* set descriptor options */
#define  F_GETFL        4       /* get descriptor options */
#define  F_SETOWN       5       /* set descriptor owner (pid/pgrp) */
#define  F_GETOWN       6       /* get descriptor owner (pid/pgrp) */
```

The F_SETFL *cmd* may be used to set a descriptor in non-blocking i/o mode and/or enable signalling when i/o is possible. F_SETOWN may be used to specify a process or process group to be signalled when using the latter mode of operation.

Operations on non-blocking descriptors will either complete immediately, note an error EWOULDBLOCK, partially complete an input or output operation returning a partial count, or return an error EINPROGRESS noting that the requested operation is in progress. A descriptor which has signalling enabled will cause the specified process and/or process group be signaled, with a SIGIO for input, output, or in-progress operation complete, or a SIGURG for exceptional conditions.

For example, when writing to a terminal using non-blocking output, the system will accept only as much data as there is buffer space for and return; when making a connection on a *socket*, the operation may return indicating that the connection establishment is "in progress". The *select* facility can be used to determine when further output is possible on the terminal, or when the connection establishment attempt is complete.

1.5.5. Descriptor wrapping.†

A user process may build descriptors of a specified type by *wrapping* a communications channel with a system supplied protocol translator:

```
new = wrap(old, proto)
result int new; int old; struct dprop *proto;
```

Operations on the descriptor *old* are then translated by the system provided protocol translator into requests on the underyling object *old* in a way defined by the protocol. The protocols supported by the kernel may vary from system to system and are described in the programmers manual.

Protocols may be based on communications multiplexing or a rights-passing style of handling multiple requests made on the same object. For instance, a protocol for implementing a file abstraction may or may not include locally generated "read-ahead" requests. A protocol that provides for read-ahead may provide higher performance but have a more difficult implementation.

Another example is the terminal driving facilities. Normally a terminal is associated with a communications line and the terminal type and standard terminal access protocol is wrapped around a synchronous communications line and given to the user. If a virtual terminal is required, the terminal driver can be wrapped around a communications link, the other end of which is held by a virtual terminal protocol interpreter.

1.6. Resource controls

1.6.1. Process priorities

The system gives CPU scheduling priority to processes that have not used CPU time recently. This tends to favor interactive processes and processes that execute only for short periods. It is possible to determine the priority currently assigned to a process, process group, or the processes of a specified user, or to alter this priority using the calls:

```
#define  PRIO_PROCESS     0     /* process */
#define  PRIO_PGRP        1     /* process group */
#define  PRIO_USER        2     /* user id */

prio = getpriority(which, who);
result int prio; int which, who;

setpriority(which, who, prio);
int which, who, prio;
```

The value *prio* is in the range -20 to 20. The default priority is 0; lower priorities cause more favorable execution. The *getpriority* call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The *setpriority* call sets the priorities of all of the specified processes to the specified value. Only the super-user may lower priorities.

† The facilities described in this section are not included in 4.2BSD.

1.6.2. Resource utilisation

The resources used by a process are returned by a *getrusage* call, returning information in a structure defined in <sys/resource.h>:

```
#define  RUSAGE_SELF        0       /* usage by this process */
#define  RUSAGE_CHILDREN    -1      /* usage by all children */

getrusage(who, rusage)
int who; result struct rusage *rusage;

struct rusage {
        struct      timeval ru_utime;   /* user time used */
        struct      timeval ru_stime;   /* system time used */
        int         ru_maxrss;          /* maximum core resident set size: kbytes */
        int         ru_ixrss;           /* integral shared memory size (kbytes*sec) */
        int         ru_idrss;           /* unshared data " */
        int         ru_isrss;           /* unshared stack " */
        int         ru_minflt;          /* page-reclaims */
        int         ru_majflt;          /* page faults */
        int         ru_nswap;           /* swaps */
        int         ru_inblock;         /* block input operations */
        int         ru_oublock;         /* block output " */
        int         ru_msgsnd;          /* messages sent */
        int         ru_msgrcv;          /* messages received */
        int         ru_nsignals;        /* signals received */
        int         ru_nvcsw;           /* voluntary context switches */
        int         ru_nivcsw;          /* involuntary " */
};
```

The *who* parameter specifies whose resource usage is to be returned. The resources used by the current process, or by all the terminated children of the current process may be requested.

1.6.3. Resource limits

The resources of a process for which limits are controlled by the kernel are defined in <sys/resource.h>, and controlled by the *getrlimit* and *setrlimit* calls:

```
#define  RLIMIT_CPU          0       /* cpu time in milliseconds */
#define  RLIMIT_FSIZE        1       /* maximum file size */
#define  RLIMIT_DATA         2       /* maximum data segment size */
#define  RLIMIT_STACK        3       /* maximum stack segment size */
#define  RLIMIT_CORE         4       /* maximum core file size */
#define  RLIMIT_RSS          5       /* maximum resident set size */

#define  RLIM_NLIMITS        6

#define  RLIM_INFINITY       0x7fffffff

struct rlimit {
        int         rlim_cur;          /* current (soft) limit */
        int         rlim_max;          /* hard limit */
};

getrlimit(resource, rlp)
int resource; result struct rlimit *rlp;

setrlimit(resource, rlp)
int resource; struct rlimit *rlp;
```

Only the super-user can raise the maximum limits. Other users may only alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*.

1.7. System operation support

Unless noted otherwise, the calls in this section are permitted only to a privileged user.

1.7.1. Bootstrap operations

The call

```
mount(blkdev, dir, ronly);
char *blkdev, *dir; int ronly;
```

extends the UNIX name space. The *mount* call specifies a block device *blkdev* containing a UNIX file system to be made available starting at *dir*. If *ronly* is set then the file system is read-only; writes to the file system will not be permitted and access times will not be updated when files are referenced. *Dir* is normally a name in the root directory.

The call

```
swapon(blkdev, size);
char *blkdev; int size;
```

specifies a device to be made available for paging and swapping.

1.7.2. Shutdown operations

The call

```
unmount(dir);
char *dir;
```

unmounts the file system mounted on *dir*. This call will succeed only if the file system is not currently being used.

The call

```
sync();
```

schedules input/output to clean all system buffer caches. (This call does not require priveleged status.)

The call

```
reboot(how)
int how;
```

causes a machine halt or reboot. The call may request a reboot by specifying *how* as RB_AUTOBOOT, or that the machine be halted with RB_HALT. These constants are defined in <sys/reboot.h>.

1.7.3. Accounting

The system optionally keeps an accounting record in a file for each process that exits on the system. The format of this record is beyond the scope of this document. The accounting may be enabled to a file *name* by doing

```
acct(path);
char *path;
```

If *path* is null, then accounting is disabled. Otherwise, the named file becomes the accounting file.

2. System facilities

This section discusses the system facilities that are not considered part of the kernel.

The system abstractions described are:

Directory contexts
: A directory context is a position in the UNIX file system name space. Operations on files and other named objects in a file system are always specified relative to such a context.

Files
: Files are used to store uninterpreted sequence of bytes on which random access *reads* and *writes* may occur. Pages from files may also be mapped into process address space. A directory may be read as a file†.

Communications domains
: A communications domain represents an interprocess communications environment, such as the communications facilities of the UNIX system, communications in the INTERNET, or the resource sharing protocols and access rights of a resource sharing system on a local network.

Sockets
: A socket is an endpoint of communication and the local point for IPC in a communications domain. Sockets may be created in pairs, or given names and used to rendezvous with other sockets in a communications domain, accepting connections from these sockets or exchanging messages with them. These operations model a labeled or unlabeled communications graph, and can be used in a wide variety of communications domains. Sockets can have different *types* to provide different semantics of communication, increasing the flexibility of the model.

Terminals and other devices
: Devices include terminals, providing input editing and interrupt generation and output flow

† Support for mapping files is not included in the 4.2 release.

control and editing, magnetic tapes, disks and other peripherals. They often support the generic *read* and *write* operations as well as a number of *ioctl*s.

Processes
Process descriptors provide facilities for control and debugging of other processes.

2.1. Generic operations

Many system abstractions support the operations *read*, *write* and *ioctl*. We describe the basics of these common primitives here. Similarly, the mechanisms whereby normally synchronous operations may occur in a non-blocking or asynchronous fashion are common to all system-defined abstractions and are described here.

2.1.1. Read and write

The *read* and *write* system calls can be applied to communications channels, files, terminals and devices. They have the form:

```
cc = read(fd, buf, nbytes);
result int cc; int fd; result caddr_t buf; int nbytes;

cc = write(fd, buf, nbytes);
result int cc; int fd; caddr_t buf; int nbytes;
```

The *read* call transfers as much data as possible from the object defined by *fd* to the buffer at address *buf* of size *nbytes*. The number of bytes transferred is returned in *cc*, which is -1 if a return occurred before any data was transferred because of an error or use of non-blocking operations.

The *write* call transfers data from the buffer to the object defined by *fd*. Depending on the type of *fd*, it is possible that the *write* call will accept some portion of the provided bytes; the user should resubmit the other bytes in a later request in this case. Error returns because of interrupted or otherwise incomplete operations are possible.

Scattering of data on input or gathering of data for output is also possible using an array of input/output vector descriptors. The type for the descriptors is defined in <sys/uio.h> as:

```
struct iovec {
        caddr_t     iov_msg;        /* base of a component */
        int         iov_len;        /* length of a component */
};
```

The calls using an array of descriptors are:

```
cc = readv(fd, iov, iovlen);
result int cc; int fd; struct iovec *iov; int iovlen;

cc = writev(fd, iov, iovlen);
result int cc; int fd; struct iovec *iov; int iovlen;
```

Here *iovlen* is the count of elements in the *iov* array.

2.1.2. Input/output control

Control operations on an object are performed by the *ioctl* operation:

```
ioctl(fd, request, buffer);
int fd, request; caddr_t buffer;
```

This operation causes the specified *request* to be performed on the object *fd*. The *request*

parameter specifies whether the argument buffer is to be read, written, read and written, or is not needed, and also the size of the buffer, as well as the request. Different descriptor types and subtypes within descriptor types may use distinct *ioctl* requests. For example, operations on terminals control flushing of input and output queues and setting of terminal parameters; operations on disks cause formatting operations to occur; operations on tapes control tape positioning.

The names for basic control operations are defined in <sys/ioctl.h>.

2.1.3. Non-blocking and asynchronous operations

A process that wishes to do non-blocking operations on one of its descriptors sets the descriptor in non-blocking mode as described in section 1.5.4. Thereafter the *read* call will return a specific EWOULDBLOCK error indication if there is no data to be *read*. The process may *declect* the associated descriptor to determine when a read is possible.

Output attempted when a descriptor can accept less than is requested will either accept some of the provided data, returning a shorter than normal length, or return an error indicating that the operation would block. More output can be performed as soon as a *select* call indicates the object is writeable.

Operations other than data input or output may be performed on a descriptor in a non-blocking fashion. These operations will return with a characteristic error indicating that they are in progress if they cannot return immediately. The descriptor may then be *selected* for *write* to find out when the operation can be retried. When *select* indicates the descriptor is writeable, a respecification of the original operation will return the result of the operation.

2.2. File system

2.2.1. Overview

The file system abstraction provides access to a hierarchical file system structure. The file system contains directories (each of which may contain other sub-directories) as well as files and references to other objects such as devices and inter-process communications sockets.

Each file is organized as a linear array of bytes. No record boundaries or system related information is present in a file. Files may be read and written in a random-access fashion. The user may read the data in a directory as though it were an ordinary file to determine the names of the contained files, but only the system may write into the directories. The file system stores only a small amount of ownership, protection and usage information with a file.

2.2.2. Naming

The file system calls take *path name* arguments. These consist of a zero or more component *file names* separated by "/" characters, where each file name is up to 255 ASCII characters excluding null and "/".

Each process always has two naming contexts: one for the root directory of the file system and one for the current working directory. These are used by the system in the filename translation process. If a path name begins with a "/", it is called a full path name and interpreted relative to the root directory context. If the path name does not begin with a "/" it is called a relative path name and interpreted relative to the current directory context.

The system limits the total length of a path name to 1024 characters.

The file name ".." in each directory refers to the parent directory of that directory. The parent directory of a file system is always the systems root directory.

The calls

```
chdir(path);
char *path;

chroot(path)
char *path;
```

change the current working directory and root directory context of a process. Only the super-user can change the root directory context of a process.

2.2.3. Creation and removal

The file system allows directories, files, special devices, and "portals" to be created and removed from the file system.

2.2.3.1. Directory creation and removal

A directory is created with the *mkdir* system call:

```
mkdir(path, mode);
char *path; int mode;
```

and removed with the *rmdir* system call:

```
rmdir(path);
char *path;
```

A directory must be empty if it is to be deleted.

2.2.3.2. File creation

Files are created with the *open* system call,

```
fd = open(path, oflag, mode);
result int fd; char *path; int oflag, mode;
```

The *path* parameter specifies the name of the file to be created. The *oflag* parameter must include O_CREAT from below to cause the file to be created. The protection for the new file is specified in *mode*. Bits for *oflag* are defined in <sys/file.h>:

```
#define  O_RDONLY      000    /* open for reading */
#define  O_WRONLY      001    /* open for writing */
#define  O_RDWR        002    /* open for read & write */
#define  O_NDELAY      004    /* non-blocking open */
#define  O_APPEND      010    /* append on each write */
#define  O_CREAT       01000  /* open with file create */
#define  O_TRUNC       02000  /* open with truncation */
#define  O_EXCL        04000  /* error on create if file exists */
```

One of O_RDONLY, O_WRONLY and O_RDWR should be specified, indicating what types of operations are desired to be performed on the open file. The operations will be checked against the user's access rights to the file before allowing the *open* to succeed. Specifying O_APPEND causes writes to automatically append to the file. The flag O_CREAT causes the file to be created if it does not exist, with the specified *mode*, owned by the current user and the group of the containing directory.

If the open specifies to create the file with O_EXCL and the file already exists, then the *open* will fail without affecting the file in any way. This provides a simple exclusive access facility.

2.2.3.3. Creating references to devices

The file system allows entries which reference peripheral devices. Peripherals are distinguished as *block* or *character* devices according by their ability to support block-oriented operations. Devices are identified by their "major" and "minor" device numbers. The major device number determines the kind of peripheral it is, while the minor device number indicates one of possibly many peripherals of that kind. Structured devices have all operations performed internally in "block" quantities while unstructured devices often have a number of special *ioctl* operations, and may have input and output performed in large units. The *mknod* call creates special entries:

```
mknod(path, mode, dev);
char *path; int mode, dev;
```

where *mode* is formed from the object type and access permissions. The parameter *dev* is a configuration dependent parameter used to identify specific character or block i/o devices.

2.2.3.4. Portal creation†

The call

```
fd = portal(name, server, param, dtype, protocol, domain, socktype)
result int fd; char *name, *server, *param; int dtype, protocol;
int domain, socktype;
```

places a *name* in the file system name space that causes connection to a server process when the name is used. The portal call returns an active portal in *fd* as though an access had occurred to activate an inactive portal, as now described.

When an inactive portal is accesseed, the system sets up a socket of the specified *socktype* in the specified communications *domain* (see section 2.3), and creates the *server* process, giving it the specified *param* as argument to help it identify the portal, and also giving it the newly created socket as descriptor number 0. The accessor of the portal will create a socket in the same *domain* and *connect* to the server. The user will then *wrap* the socket in the specified *protocol* to create an object of the required descriptor type *dtype* and proceed with the operation which was in progress before the portal was encountered.

While the server process holds the socket (which it received as *fd* from the *portal* call on descriptor 0 at activation) further references will result in connections being made to the same socket.

2.2.3.5. File, device, and portal removal

A reference to a file, special device or portal may be removed with the *unlink* call,

```
unlink(path);
char *path;
```

The caller must have write access to the directory in which the file is located for this call to be successful.

2.2.4. Reading and modifying file attributes

Detailed information about the attributes of a file may be obtained with the calls:

† The *portal* call is not implemented in 4.2BSD.

```
#include <sys/stat.h>

stat(path, stb);
char *path; result struct stat *stb;

fstat(fd, stb);
int fd; result struct stat *stb;
```

The *stat* structure includes the file type, protection, ownership, access times, size, and a count of hard links. If the file is a symbolic link, then the status of the link itself (rather than the file the link references) may be found using the *lstat* call:

```
lstat(path, stb);
char *path; result struct stat *stb;
```

Newly created files are assigned the user id of the process that created it and the group id of the directory in which it was created. The ownership of a file may be changed by either of the calls

```
chown(path, owner, group);
char *path; int owner, group;

fchown(fd, owner, group);
int fd, owner, group;
```

In addition to ownership, each file has three levels of access protection associated with it. These levels are owner relative, group relative, and global (all users and groups). Each level of access has separate indicators for read permission, write permission, and execute permission. The protection bits associated with a file may be set by either of the calls:

```
chmod(path, mode);
char *path; int mode;

fchmod(fd, mode);
int fd, mode;
```

where *mode* is a value indicating the new protection of the file. The file mode is a three digit octal number. Each digit encodes read access as 4, write access as 2 and execute access as 1, or'ed together. The 0700 bits describe owner access, the 070 bits describe the access rights for processes in the same group as the file, and the 07 bits describe the access rights for other processes.

Finally, the access and modify times on a file may be set by the call:

```
utimes(path, tvp)
char *path; struct timeval *tvp[2];
```

This is particularly useful when moving files between media, to preserve relationships between the times the file was modified.

2.2.5. Links and renaming

Links allow multiple names for a file to exist. Links exist independently of the file linked to.

Two types of links exist, *hard* links and *symbolic* links. A hard link is a reference counting mechanism that allows a file to have multiple names within the same file system. Symbolic links cause string substitution during the pathname interpretation process.

Hard links and symbolic links have different properties. A hard link insures the target file will always be accessible, even after its original directory entry is removed; no such guarantee exists for a symbolic link. Symbolic links can span file systems boundaries.

The following calls create a new link, named *path2*, to *path1*:

```
link(path1, path2);
char *path1, *path2;

symlink(path1, path2);
char *path1, *path2;
```

The *unlink* primitive may be used to remove either type of link.

If a file is a symbolic link, the "value" of the link may be read with the *readlink* call,

```
len = readlink(path, buf, bufsize);
result int len; result char *path, *buf; int bufsize;
```

This call returns, in *buf*, the null-terminated string substituted into pathnames passing through *path*.

Atomic renaming of file system resident objects is possible with the *rename* call:

```
rename(oldname, newname);
char *oldname, *newname;
```

where both *oldname* and *newname* must be in the same file system. If *newname* exists and is a directory, then it must be empty.

2.2.6. Extension and truncation

Files are created with zero length and may be extended simply by writing or appending to them. While a file is open the system maintains a pointer into the file indicating the current location in the file associated with the descriptor. This pointer may be moved about in the file in a random access fashion. To set the current offset into a file, the *lseek* call may be used,

```
oldoffset = lseek(fd, offset, type);
result off_t oldoffset; int fd; off_t offset; int type;
```

where *type* is given in <sys/file.h> as one of,

```
#define  L_SET      0    /* set absolute file offset */
#define  L_INCR     1    /* set file offset relative to current position */
#define  L_XTND     2    /* set offset relative to end-of-file */
```

The call "lseek(fd, 0, L_INCR)" returns the current offset into the file.

Files may have "holes" in them. Holes are void areas in the linear extent of the file where data has never been written. These may be created by seeking to a location in a file past the current end-of-file and writing. Holes are treated by the system as zero valued bytes.

A file may be truncated with either of the calls:

```
truncate(path, length);
char *path; int length;

ftruncate(fd, length);
int fd, length;
```

reducing the size of the specified file to *length* bytes.

2.2.7. Checking accessibility

A process running with different real and effective user ids may interrogate the accessibility of a file to the real user by using the *access* call:

```
accessible = access(path, how);
result int accessible; char *path; int how;
```

Here *how* is constructed by or'ing the following bits, defined in <sys/file.h>:

```
#define  F_OK        0      /* file exists */
#define  X_OK        1      /* file is executable */
#define  W_OK        2      /* file is writable */
#define  R_OK        4      /* file is readable */
```

The presence or absence of advisory locks does not affect the result of *access*.

2.2.8. Locking

The file system provides basic facilities that allow cooperating processes to synchronize their access to shared files. A process may place an advisory *read* or *write* lock on a file, so that other cooperating processes may avoid interfering with the process' access. This simple mechanism provides locking with file granularity. More granular locking can be built using the IPC facilities to provide a lock manager. The system does not force processes to obey the locks; they are of an advisory nature only.

Locking is performed after an *open* call by applying the *flock* primitive,

```
flock(fd, how);
int fd, how;
```

where the *how* parameter is formed from bits defined in <sys/file.h>:

```
#define  LOCK_SH     1      /* shared lock */
#define  LOCK_EX     2      /* exclusive lock */
#define  LOCK_NB     4      /* don't block when locking */
#define  LOCK_UN     8      /* unlock */
```

Successive lock calls may be used to increase or decrease the level of locking. If an object is currently locked by another process when a *flock* call is made, the caller will be blocked until the current lock owner releases the lock; this may be avoided by including LOCK_NB in the *how* parameter. Specifying LOCK_UN removes all locks associated with the descriptor. Advisory locks held by a process are automatically deleted when the process terminates.

2.2.9. Disk quotas

As an optional facility, each file system may be requested to impose limits on a user's disk usage. Two quantities are limited: the total amount of disk space which a user may allocate in a file system and the total number of files a user may create in a file system. Quotas are expressed as *hard* limits and *soft* limits. A hard limit is always imposed; if a user would exceed a hard limit, the operation which caused the resource request will fail. A soft limit results in the user receiving a warning message, but with allocation succeeding. Facilities are provided to turn soft limits into hard limits if a user has exceeded a soft limit for an unreasonable period of time.

To enable disk quotas on a file system the *setquota* call is used:

```
setquota(special, file)
char *special, *file;
```

where *special* refers to a structured device file where a mounted file system exists, and *file* refers to a disk quota file (residing on the file system associated with *special*) from which user quotas should be obtained. The format of the disk quota file is implementation dependent.

To manipulate disk quotas the *quota* call is provided:

```
#include <sys/quota.h>

quota(cmd, uid, arg, addr)
int cmd, uid, arg; caddr_t addr;
```

The indicated *cmd* is applied to the user ID *uid*. The parameters *arg* and *addr* are command specific. The file <sys/quota.h> contains definitions pertinent to the use of this call.

2.3. Interprocess communications

2.3.1. Interprocess communication primitives

2.3.1.1. Communication domains

The system provides access to an extensible set of communication *domains*. A communication domain is identified by a manifest constant defined in the file <sys/socket.h>. Important standard domains supported by the system are the "unix" domain, AF_UNIX, for communication within the system, and the "internet" domain for communication in the DARPA internet, AF_INET. Other domains can be added to the system.

2.3.1.2. Socket types and protocols

Within a domain, communication takes place between communication endpoints known as *sockets*. Each socket has the potential to exchange information with other sockets within the domain.

Each socket has an associated abstract type, which describes the semantics of communication using that socket. Properties such as reliability, ordering, and prevention of duplication of messages are determined by the type. The basic set of socket types is defined in <sys/socket.h>:

```
/* Standard socket types */
#define  SOCK_DGRAM      1     /* datagram */
#define  SOCK_STREAM     2     /* virtual circuit */
#define  SOCK_RAW        3     /* raw socket */
#define  SOCK_RDM        4     /* reliably-delivered message */
#define  SOCK_SEQPACKET  5     /* sequenced packets */
```

The SOCK_DGRAM type models the semantics of datagrams in network communication: messages may be lost or duplicated and may arrive out-of-order. The SOCK_RDM type models the semantics of reliable datagrams: messages arrive unduplicated and in-order, the sender is notified if messages are lost. The *send* and *receive* operations (described below) generate reliable/unreliable datagrams. The SOCK_STREAM type models connection-based virtual circuits: two-way byte streams with no record boundaries. The SOCK_SEQPACKET type models a connection-based, full-duplex, reliable, sequenced packet exchange; the sender is notified if messages are lost, and messages are never duplicated or presented out-of-order. Users of the last two abstractions may use the facilities for out-of-band transmission to send out-of-band data.

SOCK_RAW is used for unprocessed access to internal network layers and interfaces; it has no specific semantics.

Other socket types can be defined.†

† 4.2BSD does not support the SOCK_RDM and SOCK_SEQPACKET types.

Each socket may have a concrete *protocol* associated with it. This protocol is used within the domain to provide the semantics required by the socket type. For example, within the "internet" domain, the SOCK_DGRAM type may be implemented by the UDP user datagram protocol, and the SOCK_STREAM type may be implemented by the TCP transmission control protocol, while no standard protocols to provide SOCK_RDM or SOCK_SEQPACKET sockets exist.

2.3.1.3. Socket creation, naming and service establishment

Sockets may be *connected* or *unconnected*. An unconnected socket descriptor is obtained by the *socket* call.

```
s = socket(domain, type, protocol);
result int s; int domain, type, protocol;
```

An unconnected socket descriptor may yield a connected socket descriptor in one of two ways: either by actively connecting to another socket, or by becoming associated with a name in the communications domain and *accepting* a connection from another socket.

To accept connections, a socket must first have a binding to a name within the communications domain. Such a binding is established by a *bind* call:

```
bind(s, name, namelen);
int s; char *name; int namelen;
```

A socket's bound name may be retrieved with a *getsockname* call:

```
getsockname(s, name, namelen);
int s; result caddr_t name; result int *namelen;
```

while the peer's name can be retrieved with *getpeername*:

```
getpeername(s, name, namelen);
int s; result caddr_t name; result int *namelen;
```

Domains may support sockets with several names.

2.3.1.4. Accepting connections

Once a binding is made, it is possible to *listen* for connections:

```
listen(s, backlog);
int s, backlog;
```

The *backlog* specifies the maximum count of connections that can be simultaneously queued awaiting acceptance.

An *accept* call:

```
t = accept(s, name, anamelen);
result int t; int s; result caddr_t name; result int *anamelen;
```

returns a descriptor for a new, connected, socket from the queue of pending connections on *s*.

2.3.1.5. Making connections

An active connection to a named socket is made by the *connect* call:

```
connect(s, name, namelen);
int s; caddr_t name; int namelen;
```

It is also possible to create connected pairs of sockets without using the domain's name space to rendezvous; this is done with the *socketpair* call†:

† 4.2BSD supports *socketpair* creation only in the "unix" communication domain.

```
socketpair(d, type, protocol, sv);
int d, type, protocol; result int sv[2];
```

Here the returned *sv* descriptors correspond to those obtained with *accept* and *connect*.

The call

```
pipe(pv)
result int pv[2];
```

creates a pair of SOCK_STREAM sockets in the UNIX domain, with pv[0] only writeable and pv[1] only readable.

2.3.1.6. Sending and receiving data

Messages may be sent from a socket by:

```
cc = sendto(s, buf, len, flags, to, tolen);
result int cc; int s; caddr_t buf; int len, flags; caddr_t to; int tolen;
```

if the socket is not connected or:

```
cc = send(s, buf, len, flags);
result int cc; int s; caddr_t buf; int len, flags;
```

if the socket is connected. The corresponding receive primitives are:

```
msglen = recvfrom(s, buf, len, flags, from, fromlenaddr);
result int msglen; int s; result caddr_t buf; int len, flags;
result caddr_t from; result int *fromlenaddr;
```

and

```
msglen = recv(s, buf, len, flags);
result int msglen; int s; result caddr_t buf; int len, flags;
```

In the unconnected case, the parameters *to* and *tolen* specify the destination or source of the message, while the *from* parameter stores the source of the message, and **fromlenaddr* initially gives the size of the *from* buffer and is updated to reflect the true length of the *from* address.

All calls cause the message to be received in or sent from the message buffer of length *len* bytes, starting at address *buf*. The *flags* specify peeking at a message without reading it or sending or receiving high-priority out-of-band messages, as follows:

```
#define  MSG_PEEK        0x1    /* peek at incoming message */
#define  MSG_OOB         0x2    /* process out-of-band data */
```

2.3.1.7. Scatter/gather and exchanging access rights

It is possible scatter and gather data and to exchange access rights with messages. When either of these operations is involved, the number of parameters to the call becomes large. Thus the system defines a message header structure, in <sys/socket.h>, which can be used to conveniently contain the parameters to the calls:

```
struct msghdr {
        caddr_t     msg_name;           /* optional address */
        int         msg_namelen;        /* size of address */
        struct      iov *msg_iov;       /* scatter/gather array */
        int         msg_iovlen;         /* # elements in msg_iov */
        caddr_t     msg_accrights;      /* access rights sent/received */
        int         msg_accrightslen;   /* size of msg_accrights */
};
```

Here *msg_name* and *msg_namelen* specify the source or destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter/gather locations, as described in section 2.1.3. Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*. In the "unix" domain these are an array of integer descriptors, taken from the sending process and duplicated in the receiver.

This structure is used in the operations *sendmsg* and *recvmsg*:

```
sendmsg(s, msg, flags);
int s; struct msghdr *msg; int flags;

msglen = recvmsg(s, msg, flags);
result int msglen; int s; result struct msghdr *msg; int flags;
```

2.3.1.8. Using read and write with sockets

The normal UNIX *read* and *write* calls may be applied to connected sockets and translated into *send* and *receive* calls from or to a single area of memory and discarding any rights received. A process may operate on a virtual circuit socket, a terminal or a file with blocking or non-blocking input/output operations without distinguishing the descriptor type.

2.3.1.9. Shutting down halves of full-duplex connections

A process that has a full-duplex socket such as a virtual circuit and no longer wishes to read from or write to this socket can give the call:

```
shutdown(s, direction);
int s, direction;
```

where *direction* is 0 to not read further, 1 to not write further, or 2 to completely shut the connection down.

2.3.1.10. Socket and protocol options

Sockets, and their underlying communication protocols, may support *options*. These options may be used to manipulate implementation specific or non-standard facilities. The *getsockopt* and *setsockopt* calls are used to control options:

```
getsockopt(s, level, optname, optval, optlen)
int s, level, optname; result caddr_t optval; result int *optlen;

setsockopt(s, level, optname, optval, optlen)
int s, level, optname; caddr_t optval; int optlen;
```

The option *optname* is interpreted at the indicated protocol *level* for socket *s*. If a value is specified with *optval* and *optlen*, it is interpreted by the software operating at the specified *level*. The *level* SOL_SOCKET is reserved to indicate options maintained by the socket facilities. Other *level* values indicate a particular protocol which is to act on the option request; these values are normally interpreted as a "protocol number".

2.3.2. UNIX domain

This section describes briefly the properties of the UNIX communications domain.

2.3.2.1. Types of sockets

In the UNIX domain, the SOCK_STREAM abstraction provides pipe-like facilities, while SOCK_DGRAM provides (usually) reliable message-style communications.

2.3.2.2. Naming

Socket names are strings and may appear in the UNIX file system name space through portals†.

2.3.2.3. Access rights transmission

The ability to pass UNIX descriptors with messages in this domain allows migration of service within the system and allows user processes to be used in building system facilities.

2.3.3. INTERNET domain

This section describes briefly how the INTERNET domain is mapped to the model described in this section. More information will be found in the document describing the network implementation in 4.2BSD.

2.3.3.1. Socket types and protocols

SOCK_STREAM is supported by the INTERNET TCP protocol; SOCK_DGRAM by the UDP protocol. The SOCK_SEQPACKET has no direct INTERNET family analogue; a protocol based on one from the XEROX NS family and layered on top of IP could be implemented to fill this gap.

2.3.3.2. Socket naming

Sockets in the INTERNET domain have names composed of the 32 bit internet address, and a 16 bit port number. Options may be used to provide source routing for the address, security options, or additional address for subnets of INTERNET for which the basic 32 bit addresses are insufficient.

2.3.3.3. Access rights transmission

No access rights transmission facilities are provided in the INTERNET domain.

2.3.3.4. Raw access

The INTERNET domain allows the super-user access to the raw facilities of the various network interfaces and the various internal layers of the protocol implementation. This allows administrative and debugging functions to occur. These interfaces are modeled as SOCK_RAW sockets.

2.4. Terminals and Devices

2.4.1. Terminals

Terminals support *read* and *write* i/o operations, as well as a collection of terminal specific *ioctl* operations, to control input character editing, and output delays.

2.4.1.1. Terminal input

Terminals are handled according to the underlying communication characteristics such as baud rate and required delays, and a set of software parameters.

† The 4.2BSD implementation of the UNIX domain embeds bound sockets in the UNIX file system name space; this is a side effect of the implementation.

2.4.1.1.1. Input modes

A terminal is in one of three possible modes: *raw*, *cbreak*, or *cooked*. In raw mode all input is passed through to the reading process immediately and without interpretation. In cbreak mode, the handler interprets input only by looking for characters that cause interrupts or output flow control; all other characters are made available as in raw mode. In cooked mode, input is processed to provide standard line-oriented local editing functions, and input is presented on a line-by-line basis.

2.4.1.1.2. Interrupt characters

Interrupt characters are interpreted by the terminal handler only in cbreak and cooked modes, and cause a software interrupt to be sent to all processes in the process group associated with the terminal. Interrupt characters exist to send SIGINT and SIGQUIT signals, and to stop a process group with the SIGTSTP signal either immediately, or when all input up to the stop character has been read.

2.4.1.1.3. Line editing

When the terminal is in cooked mode, editing of an input line is performed. Editing facilities allow deletion of the previous character or word, or deletion of the current input line. In addition, a special character may be used to reprint the current input line after some number of editing operations have been applied.

Certain other characters are interpreted specially when a process is in cooked mode. The *end of line* character determines the end of an input record. The *end of file* character simulates an end of file occurrence on terminal input. Flow control is provided by *stop output* and *start output* control characters. Output may be flushed with the *flush output* character; and a *literal character* may be used to force literal input of the immediately following character in the input line.

2.4.1.2. Terminal output

On output, the terminal handler provides some simple formatting services. These include converting the carriage return character to the two character return-linefeed sequence, displaying non-graphic ASCII characters as "^character", inserting delays after certain standard control characters, expanding tabs, and providing translations for upper-case only terminals.

2.4.1.3. Terminal control operations

When a terminal is first opened it is initialized to a standard state and configured with a set of standard control, editing, and interrupt characters. A process may alter this configuration with certain control operations, specifying parameters in a standard structure:

```
struct ttymode {
        short       tt_ispeed;      /* input speed */
        int         tt_iflags;      /* input flags */
        short       tt_ospeed;      /* output speed */
        int         tt_oflags;      /* output flags */
};
```

and "special characters" are specified with the *ttychars* structure,

```
struct ttychars {
        char    tc_erasec;      /* erase char */
        char    tc_killc;       /* erase line */
        char    tc_intrc;       /* interrupt */
        char    tc_quitc;       /* quit */
        char    tc_startc;      /* start output */
        char    tc_stopc;       /* stop output */
        char    tc_eofc;        /* end-of-file */
        char    tc_brkc;        /* input delimiter (like nl) */
        char    tc_suspc;       /* stop process signal */
        char    tc_dsuspc;      /* delayed stop process signal */
        char    tc_rprntc;      /* reprint line */
        char    tc_flushc;      /* flush output (toggles) */
        char    tc_werasc;      /* word erase */
        char    tc_lnextc,      /* literal next character */
};
```

2.4.1.4. Terminal hardware support

The terminal handler allows a user to access basic hardware related functions; e.g. line speed, modem control, parity, and stop bits. A special signal, SIGHUP, is automatically sent to processes in a terminal's process group when a carrier transition is detected. This is normally associated with a user hanging up on a modem controlled terminal line.

2.4.2. Structured devices

Structures devices are typified by disks and magnetic tapes, but may represent any random-access device. The system performs read-modify-write type buffering actions on block devices to allow them to be read and written in a totally random access fashion like ordinary files. File systems are normally created in block devices.

2.4.3. Unstructured devices

Unstructured devices are those devices which do not support block structure. Familiar unstructured devices are raw communications lines (with no terminal handler), raster plotters, magnetic tape and disks unfettered by buffering and permitting large block input/output and positioning and formatting commands.

2.5. Process and kernel descriptors

The status of the facilities in this section is still under discussion. The *ptrace* facility of 4.1BSD is provided in 4.2BSD. Planned enhancements would allow a descriptor based process control facility.

II. Human/Machine Interaction and Expert Database Systems

3. Research in Human/Machine Interaction

3.1. Computer Graphics

Many applications of computer-aided geometric design require the description of objects using mathematical functions called *splines*. A spline curve is a piecewise univariate function that satisfies a set of *continuity constraints* where the curve *segments* meet. The point at which two segments join is called a *joint*. A popular type of spline is the *polynomial* spline, defined by a set of *control vertices* and a set of polynomial functions called *basis functions* that are used to blend, or weight, the vertices.

Splines are either *interpolating* or *approximating*. Interpolating splines are required to pass through the control vertices, while approximating splines are only required to pass "near" the vertices. Splines can be further classified as either *global*, or *local* representations. In a global representation, the movement of a control vertex causes the entire spline to change. In a local representation, it is possible to localize the change resulting from the perturbation of a control vertex; this is the property of *local control*. Barsky's development of the *Beta-spline*[1] has shown that it is possible to introduce *shape parameters* into the curve formulation, which can be used to modify the shape of the curve independent of the control vertices. Experience has shown that shape parameters provide a designer with intuitive control of shape.

From the standpoint of computer-aided geometric design, it is desirable to construct local, polynomial splines with shape parameters. Since the choice of interpolation versus approximation is application dependent, both should be possible. By combining the work of Catmull and Rom[2] with that of Barsky, a class of splines can be developed possessing shape parameters that are local, polynomial, and either interpolating or approximating.

Catmull and Rom introduced a class of local polynomial splines which could be made to either interpolate or approximate a set of control vertices.[1] To construct a class of splines with the properties enumerated above, we need only introduce shape parameters into the Catmull-Rom splines. As with Beta-splines, this is done by replacing *algebraic* continuity with *geometric* continuity.

Algebraic continuity refers to the continuity of parametric derivative vectors of the curve. A continuous first derivative vector gives *first order algebraic*, or C^1 continuity. If both the first and second derivative vectors are continuous, the spline has *second order algebraic* (C^2) continuity. Geometric continuity, on the other hand, requires continuity of visual quantities such as *unit tangent* and *curvature* vectors. A continuous unit tangent vector gives *first order geometric* (G^1) continuity, while *second order geometric* (G^2) continuity refers to continuous unit tangent and curvature vectors.

It had previously been shown that C^1 continuity may be replaced with G^1, and C^2 may be replaced with G^2 while still maintaining visual smoothness. Since geometric continuity is less restrictive than the corresponding order of algebraic continuity, the relaxation from algebraic to geometric continuity allows the introduction of new degrees of freedom called *shape parameters*. The replacement of C^1 with G^1 results in one shape parameter; replacing C^2 with G^2 results in two shape parameters.

We have shown[3] how the relaxation to geometric continuity can yield a class of Catmull-Rom splines, either interpolating or approximating, whose shape can be modified via shape parameters. The interpolating splines we present are new due to their shape parameters; they are

[1] Unfortunately, the title of their paper did not reflect the fact that both approximating and interpolating splines are members of the class.

the first local, polynomial, interpolating splines with locally variable shape parameters. Consequently, local modification of a shape parameter affects only a portion of the curve near the corresponding joint.

3.2. Intelligent Systems

3.2.1. The UNIX† Consultant

We have been engaged in the construction of a system called UC, for *UNIX Consultant*. UC is designed to be an automated consultant that converses in natural language with naive users to help answer their questions about the UNIX operating system. The intent is to provide a system that allows a naive user to ask questions about terminology, about command names and formats, for information concerning plans for doing things in UNIX, and for assistance in debugging problems with UNIX commands. UC should respond by explaining terminology, providing command names and describing their format, filling in the details of a user plan, suggesting plans to achieve goals, or engaging a user in a dialogue by requesting more information.

To achieve this goal requires research into basic issues in natural language processing and common sense reasoning. Our research views the user as a planning agent who has goals implicitly or explicitly expressed by his or her utterance. It is UC's task to determine the user's goal and aid the user by providing information for the achievement of those goals.

In our previous work, we had implemented a prototype version of this system. Much of our research work has addressed shortcomings with the technology that limited the success of our initial efforts.

Probably the most fundamental problem is one of knowledge representation. Weaknesses in the knowledge representation scheme we were employing, and, indeed, in knowledge representation schemes in general, prevented our system from having the flexibility and extensibility we sought. To rectify this problem, we developed a new knowledge representation scheme called KODIAK. KODIAK is described in Wilensky[4]. Some important characteristics of KODIAK are: It clarifies the semantics of slots; it is uniform, and applies to any number of semantic domains; it is based on relations, and in particular, has a small set of primitive epistemological relations and allows for the creation and definition of new relations, and it has a canonical form.

Perhaps the most interesting aspect of KODIAK is the introduction of "non-truth conditional" representation entities. We call such entities *views*. The idea behind views is that one concept can be thought of in terms of another. This viewing of another concept creates a new concept. It appears as if the introduction of such entities solves a number of long standing representational issues.

For example, one problem is that most systems provide one category called **Person**, and other called **Physical-Object**, and presume that these are in fact distinct. This is a problem because in many case, we want people to inherit properties of physical objects, even though they are not properly classified as such. We address this issue in KODIAK by asserting that **Person** is a kind of **Living-Thing**, but, in addition, we also assert that it is possible to VIEW a **Person** as a **Physical-Object**. Moreover, the VIEW of person as a **Physical-Object** is itself another concept. Namely, it is the concept **Body**. Thus we can allow **Person**. to share some of the properties of **Physical-Object** without it properly being represented as one. The importance of views is that the allow the flexibility of viewing a (possible defined) concept as something other than its "ordinary" interpretation. We believe that views constitutes a major finding in the area of knowledge representation.

Views appear to be particularly important in representing knowledge about language. Jacobs[5] points out that many otherwise unstatable linguistic regularities can be captured using views. For example, there are many cases like "John took a punch from Bill", "Bill gave John a

†UNIX is a trademark of AT&T Bell Laboratories.

punch", and "John gave Mary a kiss" in which it appears that a person being acted upon can be talked about in terms of giving and taking. Our solution is to represent a "being acted upon as transfer" view, for the purposes of linguistic expression. That is, an object being acted upon can be viewed as transferring the action to that object. Once this view is represented, knowledge about the linguistic expression of transferring (e. g., that certain transfers can be expressed using the words "give" and "take") can be employed to express certain kinds of non-transferring actions that act upon some object.

We have recently completed a second version of UC which incorporates some of these aspects of knowledge representation. KODIAK was used as the representation language for the construction of UC's knowledge-base. The knowledge-base included knowledge about linguistic patterns, UNIX concepts and commands, user and UC goals, and planning. During a course of an interaction with a user, UC adds new knowledge to the knowledge-base in the form of instantiations of the general knowledge already contained in its knowledge-base. The addition of new knowledge serves as the method of communication between the various components. Its semantics is defined by its relationship to the rest of the knowledge-base. Because it creates the form of communication between subprocesses of UC, the construction of the knowledge-base more clearly focuses the responsibilities of the various components of UC.

This version of UC can handle over 50 different question types. Among other things, it is able to generate examples where appropriate. Here is some sample actual I/O:

=> (UC)
Welcome to UC (Unix Consultant) version 3
Please type in your questions about Unix in English.
If you want information about how to use this program type 'help'.
To leave, just type 'exit'.

User: Can you tell me how to move a file?
To move a file to another file, use mv.
For example, to move the file named paul to the file named peter,
type mv paul peter.

User: Do you know how to delete a file named dekai?
To delete the file named dekai, type rm dekai.

User: Do you know how to delete a file?
To delete a file, use rm.
For example, to delete the file named lisa, type rm lisa.

User: How can I delete the file charlie?
To delete the file named charlie, type rm charlie.

User: How can I delete the file named tony?
To delete the file named tony, type rm tony.

User: How can I get rid of the file named boob?
To delete the file named boob, type rm boob.

User: I want to delete a file named peter.
To delete the file named peter, type rm peter.

User: I want to delete a file.
To delete a file, use rm.
For example, to delete the file named jim, type rm jim.

User: I want to delete the file dave.

To delete the file named dave, type rm dave.

User: How can I add read permission to a file?
To add read permission to a file, use chmod.
For example, to add individual read permission to the file named charlie, type chmod u+r charlie.

User: How can I add the owner execute permission to a file?
To add individual execute permission to a file, use chmod.
For example, to add individual execute permission to the file named marc, type chmod u+x marc.

User: How can I change the group execute permission of a file?
To change the group execute permission of a file, use chmod.
For example, to remove group execute permission from the file named jim, type chmod g-x jim.

User: How can I change the group read permission of a file?
To change the group read permission of a file, use chmod.
For example, to remove group read permission from the file named george, type chmod g-r george.

User: How can I remove the group write permission from the file named bimbo?
To remove group write permission from the file named bimbo, type chmod g-w bimbo.

User: How do I change the protection of a file?
To change the permission of a file, use chmod.
For example, to remove individual read permission from the file named paul, type chmod u-r paul.

User: How can I compact a file?
To compact a file, use compact.
For example, to compact the file named peter, type compact peter.

User: How can I create a directory?
To create a directory, use mkdir.
For example, to create the directory named paul, type mkdir paul.

User: How can I find a file?
To find a file, use whereis.
For example, to find the file named charlie, type whereis charlie.

User: How can I find out how much disk space I am using?
To find out how much disk space you are using, type du.

User: How can I find out who is using the system?
To find out all the users on the system, type finger.

User: How can I find out wilensky's phone number?
To find out Wilensky's phone number, type finger wilensky

We have also been developing additional kinds of planning mechanisms to support the reasoning component of our system. In particular, Richard Alterman has been developing a technique called *adaptive planning*[6].

The aim of adaptive planning is to borrow plans from related tasks and refit them to meet the demands of the current planning situation. For example, suppose UC has a plan for printing a file on the imagen using the UNIX 'lpr' command, by appending the argument '-Pip". An adaptive planner can re-use this plan in order to construct a plan for deleting a file from the imagen queue using the 'lprm' command by appending the argument '-Pip".

Alterman suggests several distinguishing features of adaptive planning. The first of these is that adaptive planning makes the background knowledge associated with an old plan explicit. By making the content and organization of the background knowledge explicit, it becomes possible to re-interpret the plan for a wider variety of situations. A second important feature of adaptive planning is that it plans by situation matching. Rather than treating the old plan as a partial solution which is modified using weak (problem solving) methods, the old plan is used as a starting point from which the old and new situation are matched. In the course of the matching a new plan is produced. A third important feature of adaptive planning is that it can re-use old plans in circumstances where the planner does not have access to a general plan. As shown in the example above, adaptive planning is capable of taking a specific plan for accomplishing a specific set of goals, and refit it to meet the demands of some new situation.

A prototype model of an adaptive planner is now under construction.

3.2.2. Syllogistic Reasoning in Fuzzy Logic

Fuzzy logic may be viewed as a generalization of multivalued logic in that it provides a wider range of tools for dealing with uncertainty and imprecision in knowledge representation, inference and decision analysis. In particular, fuzzy logic allows the use of (a) fuzzy predicates exemplified by *small, young, nice*, etc; (b) fuzzy quantifiers exemplified by *most, several, many, few, many more*, etc; (c) fuzzy truth-values exemplified by *quite true, very true, mostly false*, etc; (d) fuzzy probabilities exemplified by *likely, unlikely, not very likely*, etc; (e) fuzzy possibilities exemplified by *quite possible, almost impossible*, etc; and (f) predicate modifiers exemplified by *very, more or less, quite, extremely*, etc.

What matters most about fuzzy logic is its ability to deal with fuzzy quantifiers as fuzzy numbers which may be manipulated through the use of fuzzy arithmetic[7]. This ability depends in an essential way on the existence – within fuzzy logic – of the concept of cardinality or, more generally, the concept of measure of a fuzzy set. Thus, if one accepts the classical view of Kolmogoroff that probability theory is a branch of measure theory, then, more generally, the theory of fuzzy probabilities may be subsumed within fuzzy logic. This aspect of fuzzy logic makes it particularly well-suited for the management of uncertainty in expert systems[8]. More specifically, by employing a single framework for the analysis of both probabilistic and possibilistic uncertainties, fuzzy logic provides a systematic basis for inference from premises which are imprecise, incomplete or not totally reliable. In this way, it becomes possible – as we have shown – to derive a set of rules for combining evidence through conjunction, disjunction and chaining. In effect, such rules may be viewed as instances of syllogistic reasoning in fuzzy logic; however, unlike the rules employed in most of the existing expert systems, they are not *ad hoc* in nature.

A fuzzy syllogism in fuzzy logic is defined to be an inference schema in which the major premise, the minor premise and the conclusion are propositions containing fuzzy quantifiers. A basic fuzzy syllogism in fuzzy logic is the *intersection/product syllogism*

$$Q_1\ A's\ are\ B's$$

$$\underline{Q_2\ (A\ \text{and}\ B)'s\ are\ C's}$$

$$(Q_1\ dprod\ Q_2)\ A's\ are\ (B\ \text{and}\ C)'s\ ,$$

in which A, B and C are fuzzy predicates (e.g., *young men, blonde women*, etc.); Q_1 and Q_2 are fuzzy quantifiers (e.g., *most, many, almost all*, etc.) which are interpreted as fuzzy numbers; and $Q_1\ dprod\ Q_2$ is the product of Q_1 and Q_2 in fuzzy arithmetic.

We have developed several other basic syllogisms which may be employed as rules of combination of evidence in expert systems. Among these is the *consequent conjunction syllogism* which may be expressed as the inference schema

$$\frac{Q_1\ A'\text{s are } B'\text{s}}{Q_2\ A'\text{s are } C'\text{s}}$$

$$Q\ A'\text{s are } (B \text{ and } C)'\text{s} \quad ,$$

in which Q is a fuzzy number bounded from above by Q_1 *circle* Q_2 and from below by *Oorsign* (Q_1 *dsum* Q_2 *circle* 1), where *dsum* , *circle* and *circle* are the extensions of he arithmetic operators +, − and *endsign* , respectively, to fuzzy operands. Furthermore, we had shown that syllogistic reasoning in fuzzy logic provides a basis for reasoning with dispositions, that is, with propositions which are preponderantly, but not necessarily always, true.

This work may be viewed as an initiation of a study of syllogistic reasoning in the context of fuzzy logic. Such reasoning has a direct bearing on the rules of combination of evidence in expert systems and, in addition, provides a basis for inference from commonsense knowledge by viewing such knowledge as a collection of dispositions.

3.4. Software Development Systems

3.4.1. Relational Views of Programs

Making a change to a software system requires an understanding of how the part being changed fits into the system. A major part of understanding is simply seeing the information relevant to what one is trying to understand.

The OMEGA programming system[9] provides mechanisms for seeing and manipulating software in a much more powerful and general way than current systems. Instead of a linear view, such as presented by UNIX or a hierarchical view, such a presented by Gandalf[10], OMEGA provides multiple relational views of the information in a program.

The relational model provides very powerful operations for describing portions of a database of information. OMEGA gives programmers the opportunity to view and change a wide variety of cross-sections of a software system.

We have begun building a prototype implementation of OMEGA using the relational database system of INGRES[11]. So far, we have implemented a program to extract and store the information in traditional program text into an INGRES database, and a simple pointing-oriented user interface for browsing programs in the database.

Configurations, versions, call graphs, and slices are all examples of views, or cross-sections of programs. To provide a powerful mechanism for defining, retrieving, and updating these views, the OMEGA programming system uses a relational database system to manage all program information.

We have built a prototype implementation of the OMEGA - database system interface. This implementation includes the design of a relational schema for a Pascal-like language, a program for taking software stored as text and translating it into the database representation, and a simple pointing-oriented user interface. Initial performance measurements indicate that response is too slow in our current environment, but that advances in database software technology and hardware should make response fast enough in the near future.

Our initial measurements of performance show that compiled queries and buffering improve performance significantly. In general, the database system should be able to use main memory and more semantic information about the data to provide substantially better performance than is currently available.

3.3.2. A Graph Browser

The many uses of graphs in computer science provide a wide application base for a general purpose graph browser. An early prototype of such a graph browser has shown that:

- graphs should be presented in the usual fashion
- arcs and arc labels should be drawn
- convenient browsing and editing operations are required

Taking these features into account, we have proposed a design fc. GRAB[12], a general purpose graph browser. GRAB will allow users to browse graphs stored in a relational database. It will use a user-friendly window-mouse paradigm. The combination of a graphical presentation of a directed graph and the mouse as a pointing device will allow users to browse large graphs conveniently.

The database schema used by GRAB allows users to specify a map from the user's database schema so that the interface can be used for various kinds of data without much effort.

Finally, GRAB will draw graphs in a systematic form known as a proper hierarchy. In this form, graphs are partitioned into levels, with arcs directed from an upper level to the next lower level. Long arcs which traverse more than one level are shortened with the addition of dummy nodes at each intermediate level. Cycles within levels are collapsed into new nodes called proxies. Heuristics are used to minimize the arc- crossings in the graph and position the nodes on each level. The net result is a reasonable-looking graph which clearly conveys the intended entity-relationships.

The proper hierarchy layout of graphs does have a few problems. The major problem is that no provision is made for the expansion of the proxies in the graph. This problem could be solved either by expanding proxy constituents around the circumference of a circle or by precomputing a layout from proxies with a limited number of nodes. In addition, the introduction of same-level arcs to the hierarchy can reduce the number of levels and dummy nodes in the graph.

In its current implementation, the performance of the graph drawing algorithm deteriorates somewhat for large graphs. A major part of this is caused by a lack of bookkeeping in the arc minimization heuristics.

In order to improve the performance of GRAB on large graphs, some consideration will have to be given to a precomputation and incremental update scheme. This would involve precomputing the proper hierarchy and layout for a graph and updating this computation whenever the graph is changed. The tradeoff between time spent running the graph drawing algorithm anew and that spent updating an old computation has yet to be examined.

3.3.3. Machine Specific Code Improvements

The translation of a program ing language onto a target architecture requires analyses of both the language and the architecture. Much of the analysis of programming languages is now formalized and incorporated in compiler construction tools for the "front-ends" of compilers. Analysis of a target architecture by construction tools for the "back-ends" of compilers is of at least two kinds. It is sufficient to discover an implementation of each language construct on the target architecture[13]. Additional analysis may discover features of the architecture that can be exploited to generate more efficient code.

Often a target architecture contains general purpose instructions and, in addition, special purpose instructions that perform the same operations for a restricted set of operands (for example, addition versus increment). Such special purpose instructions are often faster or smaller than the equivalent more general instructions. A code generator that avoids less efficient sequences in favor of more efficient equivalent instructions produces better code. The analysis of what restrictions must hold to use special purpose instructions is tedious and prone to error if done by hand, and is susceptible to automation. Such analysis takes a suitable machine description and discovers when sequences of general purpose instructions are equivalent to special purpose instructions. One may think of the analysis as imposing a set of constraints on general purpose instructions

that make them equivalent to a special purpose instruction.

A compiler construction tool has been designed and built that automates much of the case analysis necessary to exploit special purpose instructions on a target machine. Given a suitable description of the target machine, the analysis identifies instruction sequences that are equivalent to single instructions. During code generation, these equivalences can be used to avoid inefficient sequences in favor of more efficient instructions.

A working prototype of the instruction set analyzer needed in the framework outlined by Giegerich[14] is presented by P. Kessler[15]. In contrast to the work presented in Davidson and Fraser[16] [17], machine descriptions are analyzed entirely during compiler construction (*i.e.*, once per compiler), rather than during code generation (*i.e.*, each time the compiler is used). R. Kessler[18] describes such a system for discovering equivalent instructions for instruction sequences of length 2. The techniques presented here can identify instruction sequences of arbitrary length that are equivalent to single instructions.

This analysis has been applied to the descriptions of two machines, and the results have been used to replace hand-written case analysis routines in an otherwise table driven code generator[13].

The basic idea of the analysis is to find constraints on instruction sequences so that they perform the same on the computation as a single instruction target machine. Unlike most other approaches in the literature, the sequences are found by *decomposing* single instructions to find sequences that can be constrained to be equivalent.

Previous examinations of machine descriptions for special purpose instructions have composed instruction sequences for analysis. Using composition, sequences of presumed inefficient code are constructed, and then the machine description is consulted to find a more efficient implementation of that code.

Using the composition algorithms, sequences must be composed before the machine description is examined. Thus, the number of sequences examined is an exponential function of the number of target machine instructions, whose degree is the length of the sequences considered. The composition algorithms are limited in practice to considering pairs of instructions. The composition analysis thus finds only 1-to-1 and 2-to-1 equivalences.

Instruction sequences may be extended to arbitrary lengths in the attempt to decompose an instruction. This is a major contribution of the decomposition technique. The complexity of the analysis process is exponential in the number of instructions on the target machine, with the degree of the exponential depending on the lengths of the sequences found to be equivalent. The sequences do not grow very long, since most architectures do not include extremely complex instructions (that can be decomposed by this algorithm). A performance improvement is achieved by matching the "tails" of sequences only once. In addition, trial extensions that fail (due to mismatches of the architecture or unsatisfiable constraints on the programs) are not extended further. Thus, the number of sequences considered in practice is acceptably small.

Our algorithm works from the end of the instruction towards the beginning. A more straightforward technique is to proceed forward through the instruction descriptions, using code generation techniques to discover alternative implementations. Forward decomposition is too "greedy" to find certain decompositions, however.

The descriptions of the machine include the descriptions of the computations performed by addressing modes. Thus decomposition may discover that computations implicit in operand addressing may be used to replace explicit instructions. For example a move-effective-address instruction with a source operand that adds a constant displacement to a register can be used as an alternate implementation for the addition of a constant, provided the other addend is a register.

The decomposition algorithm discovers code sequences that perform equivalent computations. The sequences are often not equivalent in cost (the difference in costs is the motivation for identifying otherwise equivalent sequences!). Both code space and execution time must be considered. Accurate costs for sequences cannot be compared during analysis at compiler

construction time, however. In part this is because instructions are analyzed for correspondence of operands, without necessarily restricting operands to particular addressing modes. Therefore, the costs for operands can not be accurately determined until a particular program is compiled. As an extreme case, the VAX-11 has equivalent code sequences where the choice of which sequence is best depends on the compile time value of an operand displacement. Thus, cost functions can not be analyzed during compiler construction.

The prototype has been used to analyze two architectures, the VAX-11 and the M68000. The analysis of the VAX-11 takes just over 2 VAX-11/750 cpu hours and discovers almost 1300 idioms. The analysis of the M68000 takes just under 4 VAX-11/750 cpu hours and discovers over 500 idioms. The longest sequences discovered were of length 3 on both architectures.

The analyzer exploits several properties of the machine descriptions to reduce the amount of work required of it. For example, families of instructions that vary only in the type of their operands can often be analyzed only once. In addition, many instructions in a target architecture can be shown to jerform unique operations, and thus there is no need to decompose them or to use them in the decomposition of other instructions.

The addition of retargetable code improvers to the suite of compiler construction tools improves the overall quality of the compilers. The uniform application of such tools provides a standard of code generator quality, which makes it possible to compare machine architectures. The availability of compilers that can exploit special purpose instructions frees machine architects to design such instructions into new machines.

3.3.4. Experiences with Code Generation for Ada

An efficient runtime representation of Ada programs was designed and implemented, using the AT&T Bell Laboratories Ada Breadboard Compiler as a foundation[19]. For lack of time the implementation did not include tasking, generic packages, or many of the implementation dependent features but did include most other features of the language. Our implementation was concerned primarily with the compiler phase starting from the high level intermediate representation DIANA and translating to the low-level IR of the portable C compiler, which is also the input language for codegen[13].

Conclusions will be presented in the following way. First, the experiences we had with DIANA and the C IR are summarized. Then the implications our runtime system design goals had on the actual BAC implementation are discussed. Next, performance measurements of the current BAC implementation are provided. With these figures we give reasons why performance data of the BAC and the ABC middle ends cannot be compared. Finally, the execution performance of Berkeley Pascal and the BAC is compared.

Conclusions about DIANA

Working with the DIANA representation was not a pleasant experience. The source reproducibility requirement of the DIANA design caused much of the DIANA structure to be unnormalised, hence more complex, larger, and less usable by the back end. In addition, DIANA is not particularly well designed for use by either the front end or the back end because important features like the symbol table are poorly represented. It is clear that the DIANA designers considered the environment tools the most important users of DIANA, and gave its space efficiency and compiler usability less consideration than they deserved.

Conclusions about the IR

Using the IR was a good decision. Because the IR provides a flexible low level representation that does not require the user to think about details such as register allocation, it is convenient and easy to use. Perhaps the greatest fault in the IR is that it is being used for more purposes than it was originally intended. The difference between the success of the IR and the failure of DIANA is clear. The IR is a form that was intended to ease retargeting C compilers to different architectures. It was not designed to be a low level representation for production quality compilers of many languages. The reason that the IR is so successful is that its value has been

established through much experience with it. From its inception DIANA was designed to be a standard for Ada intermediate representations. However, this was decided long before experience with the DIANA representation had shown one way or the other that it is a good representation. The moral is that practice with any representation is the only way to determine its true value.

Conclusions about the BAC Runtime System

The runtime system described in[18] was designed to be an efficient representation of the features necessary for a complete Ada runtime environment. Because the system shares as much descriptor information as possible, it does not provide uniform access techniques for an object's descriptor. This non-uniformity means that the compiler has to be more intelligent about the context and type of a particular object. Thus, our runtime system attempts to achieve efficient representation at the cost of greater compiler complexity.

There are some problems with this approach. The increased complexity causes the middle end to be even larger than it already has to be to implement all the features of Ada. Our implementation of the middle end is approximately 20,500 lines of C source code, including the normalization phase and IR implementation. The Ada Breadboard Compiler middle end, which has C as its target language, contained approximately 6200 lines of C[2]. Their effort was not intended to be of production quality but the size difference is still significant. Since the middle end is the most difficult part of an Ada compiler to retarget, an important part of its design should be simplicity.

Still, there are clear advantages to our runtime representation. Sharing descriptors at runtime saves considerable stack space, and saves the execution time spent initializing redundant descriptors. Using an optimizer capable of dead-variable elimination, many of the descriptors that are present but not referenced will be eliminated altogether. With this representation, any Ada type declaration which would be legal in Pascal (i.e. static arrays, and non-discriminated records) would incur no runtime overhead not also present in the Pascal implementation. In this representation, none the overhead (type descriptors, thunks, jthunks, etc.) imposes a distributed execution overhead on programs that do not use the complex features.

The Performance of the Berkeley Ada Compiler

Some useful comparisons can be made between the Berkeley Ada Compiler and the Ada Breadboard Compiler. Due to the work of Murphy, the BAC DIANA representation is much smaller than the ABC's representation. Comparative statistics for a small test program[3] are provided in Table 1 and Table 2, which were adapted from Murphy[20]. Problems arise when one tries to compare the BAC and the ABC middle ends. Because we received an early version of their middle end, which we intended to reimplement, the state of the middle end we have for comparison is incomplete. In addition, efforts at AT&T have more recently gone into building a production compiler from scratch, so figures reflecting a complete version of their middle end are impossible.

Nevertheless, in an effort to make a meaningful analysis of the quality of the code generated by the BAC middle end, comparisons will be made with *pc*, the Berkeley Unix Pascal compiler. The BAC compiles source to object code at approximately 160 lines per minute, which is 2.3 times slower than *pc*. While part of this poor performance can be attributed to the complexity of DIANA, the middle end accounts for almost a quarter of the total compile time.

Table 3 contains a comparison of execution times for 8 small benchmark programs. Perm generated all permutations of 7 objects. Towers solved the Towers of Hanoi. Queens solved the 8 queens problem. IntMM and MM did an integer and real matrix multiply, respectively. Puzzle has been introduced. Quick, Bubble, and Tree were all sorting algorithms. FFT solved a fast

[2] This figure is somewhat suspect. See the discussion in the following section.

[3] The program was the ubiquitous Puzzle program of Forest Baskett, upon which seemingly thousands of analyses have unfortunately been based.

Puzzle Space Requirements		
246 lines of Ada source 1635 input tokens (8 tokens/line) 3218 DIANA nodes (16 nodes/line) 1834 abstract syntax tree nodes 921 symbol table nodes 3073 sequences (list elements)		
	ABC	BAC
size of DIANA	254,816	140,276 bytes
average node size	72	35 bytes
external DIANA file	232	140 Kbytes

Table 1: Compilation Space Requirements for the Puzzle Program

Puzzle Compile Times on a VAX 11/750		
	ABC	BAC
front end	28.3	47.3 seconds
middle end	-	20.7 seconds
back end	-	24.8 seconds
total	-	92.8 seconds

Table 2: Compilation Execution for the Puzzle Program

Program	Berkeley Pascal (sec)	Berkeley Ada (sec)
Perm	2.7	3.6
Towers	2.8	3.5
Queens	1.6	1.0
IntMM	2.2	2.0
MM	2.7	2.4
Puzzle	12.9	8.7
Quick	1.7	1.8
Bubble	3.0	2.0
Tree	6.4	4.6
FFT	4.8	4.3
Total	40.8	33.9

Table 3. Comparison of Pascal and Ada Execution Times

Fourier transform. Because the programs are so small, the significance of the comparison is questionable, but larger programs written in both Pascal and Ada are not available. The table shows the execution performance of the two compilers is comparable. The worst case for the BAC implementation was Perm, which suffered because it called a 3 line assembler function called

Swap repeatedly. The overhead of passing the static link to the callee, and setting up the static linkage was the cause of the poor performance. A suggested modification of the simple static linkage model presented here would solve the problem, but also considerably complicate the model.

We view the favorable comparison with a language as relatively simple as Pascal as evidence that our runtime implementation was successful. In conclusion, we feel that the runtime system suggested provides efficient execution time performance with little distributed overhead, and offers a conceptually simple and useful runtime model for Ada.

3.3.5. Smalltalk Implementation Techniques for a RISC Architecture

There are several reasons why Smalltalk programs have proven especially difficult to execute quickly.

- The language has been defined in terms of a bytecode interpreter. Interpreters are slow.

- The pure object-orientation of the language implies a huge number of procedure calls ("messages"), which are often time-consuming in conventional implementations.

- The definition of Smalltalk execution and the style of its customary use require the rapid creation and automatic reclamation of many objects. This puts a heavy demand on the memory management mechanism.

We have implemented Smalltalk-80 on an instruction-level simulator for a RISC microcomputer called SOAR. Measurements suggest that even a conventional computer can provide high performance for Smalltalk-80 by

- abandoning the 'Smalltalk Virtual Machine' in favor of compiling Smalltalk directly to SOAR machine code,

- compiling Smalltalk directly to SOAR machine code,

- linearizing the activation records on the machine stack,

- eliminating the object table, and

- replacing reference counting with a new technique called Generation Scavenging. In order to implement these techniques, we had to find new ways of

- hashing objects,

- accessing often-used objects, invoking blocks,

- referencing activation records,

- managing activation record stacks, and

- converting the virtual machine images.

These techniques have been summarized by Samples et. al[21].

3.3.6. The PAN Language-Based Editor

Pan is a multilingual language-based editor for manipulating tree-structured documents. The editor supports both tree- and text-oriented operations. The expected use of this system is as the front-end for a development environment in which experienced developers use several languages while creating a complex program or other document. One task of the front-end is to

gather and make available information about the document for use by the developers and by other tools.

Multiple languages are handled by separating the language-specific information from the generic utilities supplied by the editor. Language-specific information, in the form of a language description, is preprocessed into tables for use by the editor. The editing component itself is table-driven. New languages can be added to the system by creating and loading a new set of tables. Pan is designed to handle different languages in different editing workspaces; switching workspaces within an editing session allows the user to edit different languages.

There are two major components to the Pan system: the editor and the table generator. The editor supplies editing operations while checking that the document meets the requirements of the language in which it is written. These requirements fall into three categories: lexical, syntactic, and contextual. (Contextual requirements are often called the "static semantics" of a language.) Information concerning errors or inconsistencies in the document is communicated to the user during the course of editing.

The editor uses both the concrete representation of a document (the representation as seen by a user of the system) and the abstract syntax of the document to implement its editing operations. The correspondence between the two representations is maintained by an incremental scanning and parsing system. The abstract syntax is in the form of an operator/phylum tree[22]. Contextual constraints are enforced using only the abstract syntax. Other tools in the environment may add information to the internal tree representation; it is the structure of the tree which is of primary interest to the editor.

The table generator takes a language description, checks it for consistency and for the properties required by the algorithms used in Pan, and then generates the tables used by the editor. In fact, the table generator is a collection of tools, many of which already exist in the UNIX programming environment.

Pan will be implemented on a SUN workstation.[4] The primary implementation language will be LISP, with recourse to C for low level routines and access to the screen.

3.3.7. The VorTeX System

The goal of the Berkeley VorTeX[5] project is to build a new document processing environment with the following major features:

(1) Interactive: The system allows the user to edit/preview TeX documents interactively. In order to achieve a satisfactory degree of interaction the system must be incremental (as opposed to batch) in both reformatting and redisplay; namely it only reprocesses those portions of the document that have been changed during an editing/previewing session.

(2) Multiple Representations: Both source (ASCII representation) and proof (bitmap representation) of the document are maintained by the system, with, of course, an intermediate representation transparent to the user. The user is allowed to edit both representations, whichever he/she considers more convenient. Most importantly, changes made to one representation must be propagated to the other automatically.

(3) Extensible: The system must be flexible enough to incorporate objects such as program fragments, tables, graphics, ..., etc. In particular, TeX's powerful macro facility must be preserved.

(4) Easy to use and reasonably portable.

---

[4] SUN workstation is a trademark of Sun Microsystems, Inc.

[5] For Visually Oriented TeX.

Based on the design goals listed above, the following requirements have been identified:

(1) VorTeX and TeX must be able to generate identical outputs.

(2) The user should feel more comfortable with VorTeX than with the current disintegrated environment.

VorTeX's fundamental departure from other systems is in the notion of multiple representations. Being interactive and incremental is nothing new in the field of text processing. A number of so-called WYSWYG (What You See is What You Get) programs have emerged in the past few years that advocate friendly user interfaces. Unfortunately, such nice user interfaces often turn out to be discriminating against professional and experienced users — power and ease of use sometimes contradict each other. Furthermore, of the many WYSWYG systems available, none seem to be able to produce documents of higher quality than that of TeX, a batch-oriented and far less interactive system.

It is our belief that having a good user interface is important, but not at the cost of the quality and flexibility of TeX. The ideal scenario is that the user be allowed to modify the appearance of the document as in any WYSWYG systems and that he/she be able to enter high-level formatting commands if that is considered more convenient. The implication is that both source and proof of the document must be available in the environment, and the corresponding representation coherence problem becomes central to this research. This multiple representation problem is not unique to document processing *per se*. Almost all large-scale software systems such as VLSI, CAD, programming environments, and databases have similar problems.

We propose an object-oriented approach to this whole environment. The smallest object of granularity is, in TeX's jargon, an "hbox". An object contains pointers to its positions in both representations, pointers its neighboring objects, and (implicit) pointers to the class it inherits. It also maintains some format attributes such as font type, size, ..., etc. and the information for handling representation coherence, i.e. a declarative set of legitimate bitmap operations with respect to this particular object type and the corresponding TeX commands. Changes made to an object are broadcast to related parties. Higher order objects such as paragraphs and pages have access to the various formatting and displaying modules and based on some heuristics decide when to incrementally reprocess the document. The advantages of this approach are:

(1) Open system: new objects and modules can be plugged in easily.

(2) Incremental: message passing triggers the necessary updates at some reasonable granularity.

Work on VorTeX was begun under this contract and is continuing under the new contract. In addition to developing the overall design of the project, we have obtained the results described in the following sections.

Fonts for TeX

In the process of providing fonts for the various output devices that are available to produce TeX output, we have encountered many situations that require non-standard fonts. One good example of this is providing fonts for the Sun workstation preview program dvitool. At the low resolution of the screen, it is difficult to create good looking fonts, and most fonts require tuning to be even readable.

There has always been a difficulty in tuning existing fonts because the files are binary and cannot be easily modified with the existing editors. Previously, the only way to do this was by converting the font into an ascii file with the character bitmaps represented as an array of asterisks and periods, which could be edited. After the file was changed, this ascii representation could be converted back to the binary font format. Needless to say, this was a cumbersome way to make changes and only relatively small fonts could be dealt with. Also, creating new fonts was not easily done because of the format of the ascii representation files.

To help in modifying the binary fonts, we have written pxtool which is a Sun Windows bitmap editor for the pixel fonts. It takes a font file and allows one to change the pixels in the font on the screen. The format of the tool is something like the standard bitmap editor IconTool provided by Sun. The character image is actually changed by clicking mouse keys at the appropriate pixels and font parameters may be changed by typing them to prompts at the top of the tool.

DVItool: TeX Without Paper

DVItool is a previewer for TeX that operates on the SUN family of computers under the SUN window system. It is an offspring in a long line of DVI-to-whatever programs which differs in one fundamental aspect; it is run in its own window concurrently with an arbitrary number of other processes. DVItool is meant to be run once during the entire computing session rather than executed each time a DVI file is to be previewed. Concurrency has mandated that DVItool be robust enough in its error handling and flexible enough in its command structure to be useful throughout a potentially lengthy session.

A number of features have been added to DVItool to make it a powerful and flexible tool: the ability to change directories, a single keystroke command to re-read a potentially updated DVI file, the ability to do wild-card page searches on any of TeX's ten count variables, an initialization file to allow a degree of user customization, and six levels of magnification corresponding to TeX's six magsteps.

In addition, another program, TeXdvi, has been written to simplify the edit-TeX-preview cycle. The standard method of creating TeX documents is to edit the TeX document with one's favorite text editor, run TeX on the edited document, and then send the resulting DVI file to either a previewer or a hard-copy device. TeXdvi simplifies this cycle by consolidating the last two steps: it runs TeX[6] on the TeX source and then sends DVItool a message to preview the resulting DVI file. If there are errors in the TeX job, TeXdvi queries the user about whether to preview the DVI file.

The combination of DVItool and TeXdvi alleviates much of the tedium associated with the creation of documents using a batch-oriented text processor such as TeX.

4. Research in Expert Database Systems

4.1. Introduction

Under this contract we have developed an extended language, QUEL*, containing the following capabilities:

the ability to support procedural data and the capability to execute it
the ability to reference subobjects within procedurally defined objects
the ability to perform indefinite iteration

We have also built an optimizer for this language, and performed initial exploration on how to generalize the language to provide a rules system. These three topics are discussed in the following sections.

4.2. The Definition of QUEL*

Basically, any column of a relation in QUEL* can hold a data element that is a procedure. For example, the following tuple contains data on an employee indicating his name, salary, manager, age, and the department that he works in. The last data object is defined procedurally.

name	salary	manager	age	dept
Mike	2000	Fred	25	retrieve (DEPT.all) where DEPT.name = "shoe"

[6] or any other TeX processor such as LaTeX or SliTeX.

An EMP Tuple
Table 1

Procedural data can be executed as follows:

execute (EMP.dept) where EMP.name = "Mike"

Moreover, if a DEPT object has fields dname and floor, then one can find the floor number that Mike works on by running the following command.

retrieve (EMP.DEPT.floor) where EMP.name = "Mike"

Here, we use a nested dot notation, e.g. EMP.DEPT.floor, to reference a subject within the procedurally defined object DEPT. This addressing was pioneered by the data sublanguage, GEM[23]. Lastly, indefinite iteration of commands are supported by including a * operator. Hence, we can update the salaries of all employees to be no larger than that of their direct manager or anyone else that they report to indirectly as follows:

replace* EMP (salary = M.salary) from M in EMP
where EMP.manager = M.name
and EMP.salary > M.salary

The * indicates that logically the command should continue to execute until it no longer has any effect.

4.3. Optimization of QUEL*

We have spent considerable time designing an optimizer for QUEL* and in extending the prototype INGRES relational database system[24] to optimize and execute QUEL* commands. We illustrate the design of the optimizer with an extended example in this section. The full report on the design and implementation of QUEL* has been accepted for publication in the ACM Transactions on Database Systems[25].

Although more sophisticated query processing algorithms have been constructed[26] [27] our implementation builds on the original INGRES strategy[28]. The implementation of QUEL* has been accomplished using this code because it is readily available for experimentation. Integration of our constructs into more advanced optimizers appears straightforward, and we discuss this point in more detail at the end of this section.

Detachment of one-variable queries that do not contain multiple dot or relation level operators can proceed as in the original INGRES algorithms[28]. Similarly, the reduction module of decomposition is unaffected by our extensions to QUEL. In addition, tuple substitution is performed when all other processing steps fail. A glance at the left hand column of Figure 1 indicates that a test for zero variables must be inserted into the original flow of control after the reduction module. Then, new facilities must be included to process the "yes" branch of the test. These include a test for whether there is a relation to materialize and the code to perform this step. Lastly, the one-variable query processor must be extended to process relation level operators. We explain these extensions with a detailed example.

The desired task is to find the polygon descriptions with identifiers less than 5 for all objects which have the same collection of shapes as the complex object with Oid equal to 10, i.e:

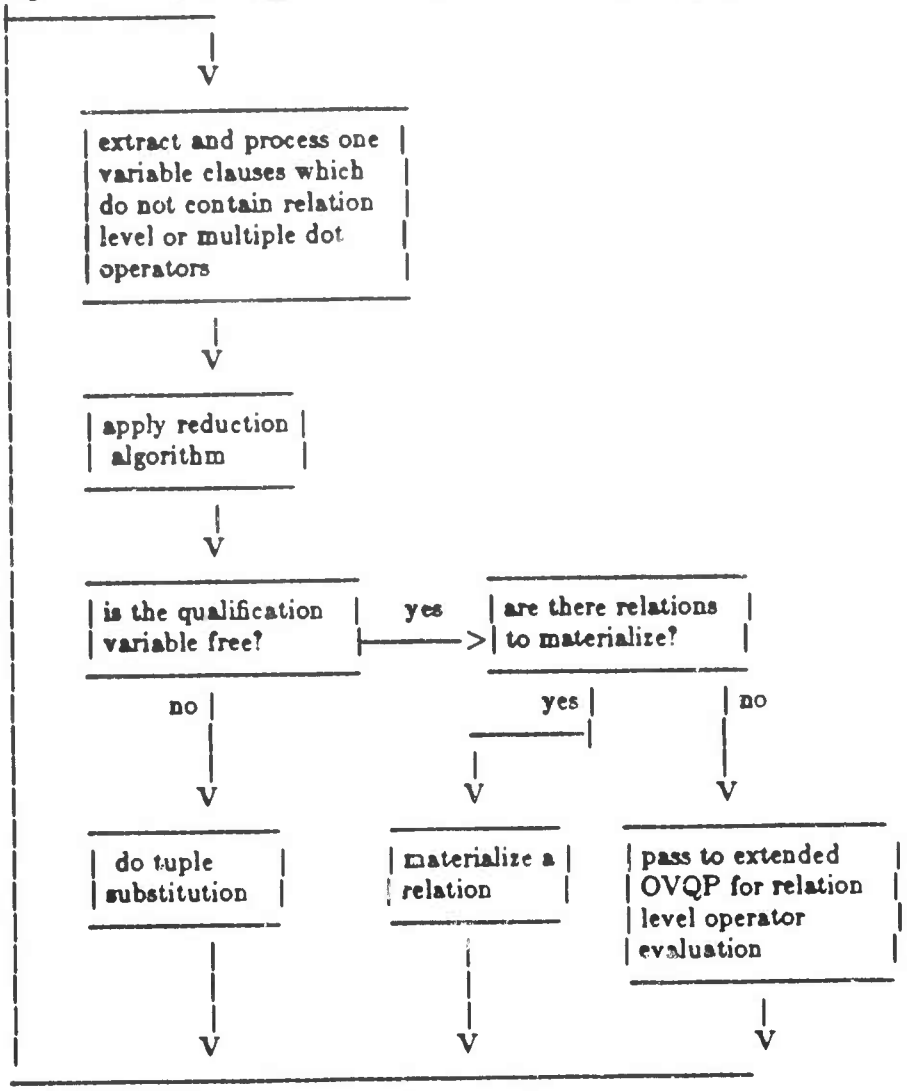
range of o is OBJECT
range of o1 is OBJECT
retrieve (o.shape.p-desc)
    where o.shape.Pid < 5
    and o.shape == o1.shape
    and o1.Oid = 10

The initial step of the reduction process finds that the final clause in the query has only a single variable, so it can be executed as:

retrieve into TEMP-1 (o1.shape) where o1.Oid = 10

The original query is now:

Figure 1 shows a diagram of the extended decomposition proc s.



Extended Decomposition

Figure 1

```
retrieve (o.shape.p-desc)
        where o.shape.Pid < 5
        and o.shape == TEMP-1.shape
```

The first clause above contains a multiple dot attribute and should not be processed until later. At this point reduction fails and the query still has two variables in it, so processing falls through to the tuple substitution module. If TEMP-1 is selected for substitution, the resulting query is:

```
retrieve (o.shape.p-desc)
        where o.shape.Pid < 5
        and o.shape == "QUEL-constant-1"
```

Notice that the variable "TEMP-1.shape" has been replaced by a constant (QUEL-constant-1) which is a collection of QUEL commands. Processing now returns to the top of the loop where the query still does not have any one-variable clauses. Processing again returns to tuple

substitution where the variable o is chosen. This results in the query:

```
retrieve ("QUEL-constant-2".p-desc)
        where "QUEL-constant-2".Pid < 5
        and "QUEL-constant-3" == "QUEL-constant-1"
```

Notice that o.shape has been replaced by two constants (QUEL-constant-2 and QUEL-constant-3) which are identical. When o.shape is materialized, there will be a one-relation clause (o.shape.Pid < 5) that can be used to restrict and project the relation. Moreover, it is desirable to check this clause as early as possible because the current query will have no answer if this clause is false. On the other hand, o.shape must be retained as a complete object so that the the relation level comparison with QUEL-constant-1 can be performed if necessary. In order to avoid forcing the relation level operator to be executed first, we have duplicated the QUEL constant and thereby retained the option of performing the one-variable restriction first. Even though QUEL-constant-2 and QUEL-constant-3 define the same object, the caching module contained in our prototype should avoid materializing this object more than once.

Now the command has zero variables and is passed to the materialize module. This processing step chooses one of the QUEL constants and materializes the outer-union of the RETRIEVE and DEFINE VIEW commands into a relation TEMP-2. If "QUEL-constant-2" is chosen, then the resulting query will be:

```
retrieve (TEMP-2.p-desc) where
        "QUEL-constant-3" == "QUEL-constant-1"
        and TEMP-2.Pid < 5
```

This query now has a one-variable clause which can be detached and processed creating another temporary relation TEMP-3. If TEMP-3 is empty then the query is false and can be terminated. Alternately, processing must continue on the following command:

```
retrieve (TEMP-3.p-desc) where
        "QUEL-constant-3" == "QUEL-constant-1"
```

The qualification is again free from variables, so another relation must be materialized. If "QUEL-constant-1" is chosen, we obtain:

```
retrieve (TEMP-3.p-desc) where
        "QUEL-constant-3" == TEMP-4
```

The qualification is still free from variables, so the final relation must be materialized as follows:

```
retrieve (TEMP-3.p-desc) where
        TEMP-5 == TEMP-4
```

After another trip around the processing loop, no further materialization is possible. Hence, the query must now be passed to the one-variable query processor. This module will process the operator == for the two relations involved.

Several comments are appropriate at this time. This algorithm delays materializing a relation until there is no conventional processing to do. In addition, it delays evaluating relation level operators until there is nothing else to do. This reflects our belief that expensive operations should never be done until absolutely necessary.

Second, most current optimizers build a complete query plan in advance of executing the command. Such optimizers (e.g. [25]) can construct a plan for the portion of the query without nested dot constructs. However, run-time planning may be required on remaining portions of commands. For example, the following query must be processed by tuple substitution for o or o1.

```
retrieve (o.shape.p-desc, o1.shape.p-desc)
where o.shape.l-desc = o1.shape.l.desc
```

After substitution twice, the remaining query is:

```
retrieve (TEMP-1.p-desc, TEMP-2.p-desc)
where TEMP-1.l-desc = TEMP-2.l-desc
```

The characteristics of TEMP-1 and TEMP-2 are not known until run time, so further query planning must be deferred to this time.

Lastly, in our prototype the module that materializes a relation passes the RETRIEVE commands to another process which also runs the INGRES code. This second INGRES executes the command, stores the resulting relation in the database, and then passes control back to the first INGRES. A second process is required because the INGRES code will not allow a command to suspend in the middle of the decomposition process so that a new command can be executed. The ability to "stack" the execution state of a query would be a very desirable addition to the system.

Appendix A – Summary of facilities

1. Kernel primitives

1.1. Process naming and protection

sethostid	set UNIX host id
gethostid	get UNIX host id
sethostname	set UNIX host name
gethostname	get UNIX host name
getpid	get process id
fork	create new process
exit	terminate a process
execve	execute a different process
getuid	get user id
geteuid	get effective user id
setreuid	set real and effective user id's
getgid	get accounting group id
getegid	get effective accounting group id
getgroups	get access group set
setregid	set real and effective group id's
setgroups	set access group set
getpgrp	get process group
setpgrp	set process group

1.2 Memory management

<mman.h>	memory management definitions
sbrk	change data section size
sstk†	change stack section size
getpagesize	get memory page size
mmap†	map pages of memory
mremap†	remap pages in memory
munmap†	unmap memory
mprotect†	change protection of pages
madvise†	give memory management advice
mincore†	determine core residency of pages

1.3 Signals

<signal.h>	signal definitions
sigvec	set handler for signal
kill	send signal to process
killpgrp	send signal to process group
sigblock	block set of signals
sigsetmask	restore set of blocked signals
sigpause	wait for signals
sigstack	set software stack for signals

1.4 Timing and statistics

<sys/time.h>	time-related definitions
gettimeofday	get current time and timezone
settimeofday	set current time and timezone
getitimer	read an interval timer
setitimer	get and set an interval timer

† Not supported in 4.2BSD.

profil	profile process

1.5 Descriptors

getdtablesize	descriptor reference table size
dup	duplicate descriptor
dup2	duplicate to specified index
close	close descriptor
select	multiplex input/output
fcntl	control descriptor options
wrap†	wrap descriptor with protocol

1.6 Resource controls

<sys/resource.h>	resource-related definitions
getpriority	get process priority
setpriority	set process priority
getrusage	get resource usage
getrlimit	get resource limitations
setrlimit	set resource limitations

1.7 System operation support

mount	mount a device file system
swapon	add a swap device
umount	umount a file system
sync	flush system caches
reboot	reboot a machine
acct	specify accounting file

2. System facilities

2.1 Generic operations

read	read data
write	write data
<sys/uio.h>	scatter-gather related definitions
readv	scattered data input
writev	gathered data output
<sys/ioctl.h>	standard control operations
ioctl	device control operation

2.2 File system

Operations marked with a * exist in two forms: as shown, operating on a file name, and operating on a file descriptor, when the name is preceded with a "f".

<sys/file.h>	file system definitions
chdir	change directory
chroot	change root directory
mkdir	make a directory
rmdir	remove a directory
open	open a new or existing file
mknod	make a special file
portal†	make a portal entry
unlink	remove a link
stat*	return status for a file

† Not supported in 4.2BSD.

lstat	returned status of link
chown*	change owner
chmod*	change mode
utimes	change access/modify times
link	make a hard link
symlink	make a symbolic link
readlink	read contents of symbolic link
rename	change name of file
lseek	reposition within file
truncate*	truncate file
access	determine accessibility
flock	lock a file

2.3 Communications

<sys/socket.h>	standard definitions
socket	create socket
bind	bind socket to name
getsockname	get socket name
listen	allow queueing of connections
accept	accept a connection
connect	connect to peer socket
socketpair	create pair of connected sockets
sendto	send data to named socket
send	send data to connected socket
recvfrom	receive data on unconnected socket
recv	receive data on connected socket
sendmsg	send gathered data and/or rights
recvmsg	receive scattered data and/or rights
shutdown	partially close full-duplex connection
getsockopt	get socket option
setsockopt	set socket option

2.4 Terminals, block and character devices

2.5 Processes and kernel hooks

Appendix B – File System Implementation

1. Introduction

This appendix describes the changes from the original 512 byte UNIX file system to the new one released with the 4.2 Berkeley Software Distribution. It presents the motivations for the changes, the methods used to affect these changes, the rationale behind the design decisions, and a description of the new implementation. This discussion is followed by a summary of the results that have been obtained, directions for future work, and the additions and changes that have been made to the user visible facilities. The paper concludes with a history of the software engineering of the project.

The original UNIX system that runs on the PDP-11† has simple and elegant file system facilities. File system input/output is buffered by the kernel; there are no alignment constraints on data transfers and all operations are made to appear synchronous. All transfers to the disk are in 512 byte blocks, which can be placed arbitrarily within the data area of the file system. No constraints other than available disk space are placed on file growth [Ritchie74], [Thompson79].

When used on the VAX-11 together with other UNIX enhancements, the original 512 byte UNIX file system is incapable of providing the data throughput rates that many applications require. For example, applications that need to do a small amount of processing on a large quantities of data such as VLSI design and image processing, need to have a high throughput from the file system. High throughput rates are also needed by programs with large address spaces that are constructed by mapping files from the file system into virtual memory. Paging data in and out of the file system is likely to occur frequently. This requires a file system providing higher bandwidth than the original 512 byte UNIX one which provides only about two percent of the maximum disk bandwidth or about 20 kilobytes per second per arm [White80], [Smith81b].

Modifications have been made to the UNIX file system to improve its performance. Since the UNIX file system interface is well understood and not inherently slow, this development retained the abstraction and simply changed the underlying implementation to increase its throughput. Consequently users of the system have not been faced with massive software conversion.

Problems with file system performance have been dealt with extensively in the literature; see [Smith81a] for a survey. The UNIX operating system drew many of its ideas from Multics, a large, high performance operating system [Feiertag71]. Other work includes Hydra [Almes78], Spice [Thompson80], and a file system for a lisp environment [Symbolics81a].

A major goal of this project has been to build a file system that is extensible into a networked environment [Holler73]. Other work on network file systems describe centralized file servers [Accetta80], distributed file servers [Dion80], [Luniewski77], [Porcar82], and protocols to reduce the amount of information that must be transferred across a network [Symbolics81b], [Sturgis80].

2. Old File System

In the old file system developed at Bell Laboratories each disk drive contains one or more file systems.† A file system is described by its super-block, which contains the basic parameters of the file system. These include the number of data blocks in the file system, a count of the maximum number of files, and a pointer to a list of free blocks. All the free blocks in the system are chained together in a linked list. Within the file system are files. Certain files are distinguished as directories and contain pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. The inode contains information describing

† DEC, PDP, VAX, MASSBUS, and UNIBUS are trademarks of Digital Equipment Corporation.

† A file system always resides on a single drive.

ownership of the file, time stamps marking last modification and access times for the file, and an array of indices that point to the data blocks for the file. For the purposes of this section, we assume that the first 8 blocks of the file are directly referenced by values stored in the inode structure itself*. The inode structure may also contain references to indirect blocks containing further data block indices. In a file system with a 512 byte block size, a singly indirect block contains 128 further block addresses, a doubly indirect block contains 128 addresses of further single indirect blocks, and a triply indirect block contains 128 addresses of further doubly indirect blocks.

A traditional 150 megabyte UNIX file system consists of 4 megabytes of inodes followed by 146 megabytes of data. This organization segregates the inode information from the data; thus accessing a file normally incurs a long seek from its inode to its data. Files in a single directory are not typically allocated slots in consecutive locations in the 4 megabytes of inodes, causing many non-consecutive blocks to be accessed when executing operations on all the files in a directory.

The allocation of data blocks to files is also suboptimum. The traditional file system never transfers more than 512 bytes per disk transaction and often finds that the next sequential data block is not on the same cylinder, forcing seeks between 512 byte transfers. The combination of the small block size, limited read-ahead in the system, and many seeks severely limits file system throughput.

The first work at Berkeley on the UNIX file system attempted to improve both reliability and throughput. The reliability was improved by changing the file system so that all modifications of critical information were staged so that they could either be completed or repaired cleanly by a program after a crash [Kowalski78]. The file system performance was improved by a factor of more than two by changing the basic block size from 512 to 1024 bytes. The increase was because of two factors; each disk transfer accessed twice as much data, and most files could be described without need to access through any indirect blocks since the direct blocks contained twice as much data. The file system with these changes will henceforth be referred to as the *old file system*.

This performance improvement gave a strong indication that increasing the block size was a good method for improving throughput. Although the throughput had doubled, the old file system was still using only about four percent of the disk bandwidth. The main problem was that although the free list was initially ordered for optimal access, it quickly became scrambled as files were created and removed. Eventually the free list became entirely random causing files to have their blocks allocated randomly over the disk. This forced the disk to seek before every block access. Although old file systems provided transfer rates of up to 175 kilobytes per second when they were first created, this rate deteriorated to 30 kilobytes per second after a few weeks of moderate use because of randomization of their free block list. There was no way of restoring the performance an old file system except to dump, rebuild, and restore the file system. Another possibility would be to have a process that periodically reorganized the data on the disk to restore locality as suggested by [Maruyama76].

3. New file system organization

As in the old file system organization each disk drive contains one or more file systems. A file system is described by its super-block, that is located at the beginning of its disk partition. Because the super-block contains critical data it is replicated to protect against catastrophic loss. This is done at the time that the file system is created; since the super-block data does not change, the copies need not be referenced unless a head crash or other hard disk error causes the default super-block to be unusable.

* The actual number may vary from system to system, but is usually in the range 5-13.

To insure that it is possible to create files as large as 2†32 bytes with only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of the file system is maintained in the super-block so it is possible for file systems with different block sizes to be accessible simultaneously on the same system. The block size must be decided at the time that the file system is created; it cannot be subsequently changed without rebuilding the file system.

The new file system organization partitions the disk into one or more areas called *cylinder groups*. A cylinder group is comprised of one or more consecutive cylinders on a disk. Associated with each cylinder group is some bookkeeping information that includes a redundant copy of the super-block, space for inodes, a bit map describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. For each cylinder group a static number of inodes is allocated at file system creation time. The current policy is to allocate one inode for each 2048 bytes of disk space, expecting this to be far more than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. Thus a single hardware failure that destroyed the top platter could cause the loss of all copies of the redundant super-blocks. Thus the cylinder group bookkeeping information begins at a floating offset from the beginning of the cylinder group. The offset for each successive cylinder group is calculated to be about one track further from the beginning of the cylinder group. In this way the redundant information spirals down into the pack so that any single track, cylinder, or platter can be lost without losing all copies of the super-blocks. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information is used for data blocks.†

3.1. Optimizing storage utilization

Data is laid out so that larger blocks can be transferred in a single disk transfer, greatly increasing file system throughput. As an example, consider a file in the new file system composed of 4096 byte data blocks. In the old file system this file would be composed of 1024 byte blocks. By increasing the block size, disk accesses in the new file system may transfer up to four times as much information per disk transaction. In large files, several 4096 byte blocks may be allocated from the same cylinder so that even larger data transfers are possible before initiating a seek.

The main problem with bigger blocks is that most UNIX file systems are composed of many small files. A uniformly large block size wastes space. Table 1 shows the effect of file system block size on the amount of wasted space in the file system. The machine measured to obtain these figures is one of our time sharing systems that has roughly 1.2 Gigabyte of on-line storage. The measurements are based on the active user file systems containing about 920 megabytes of formated space.

Space used	% waste	Organization
775.2 Mb	0.0	Data only, no separation between files
807.8 Mb	4.2	Data only, each file starts on 512 byte boundary
828.7 Mb	6.9	512 byte block UNIX file system
866.5 Mb	11.8	1024 byte block UNIX file system
948.5 Mb	22.4	2048 byte block UNIX file system
1128.3 Mb	45.6	4096 byte block UNIX file system

Table 1 - Amount of wasted space as a function of block size.

† While it appears that the first cylinder group could be laid out with its super-block at the "known" location, this would not work for file systems with blocks sizes of 16K or greater, because of the requirement that the cylinder group information must begin at a block boundary.

The space wasted is measured as the percentage of space on the disk not containing user data. As the block size on the disk increases, the waste rises quickly, to an intolerable 45.6% waste with 4096 byte file system blocks.

To be able to use large blocks without undue waste, small files must be stored in a more efficient way. The new file system accomplishes this goal by allowing the division of a single file system block into one or more *fragments*. The file system fragment size is specified at the time that the file system is created; each file system block can be optionally broken into 2, 4, or 8 fragments, each of which is addressable. The lower bound on the size of these fragments is constrained by the disk sector size, typically 512 bytes. The block map associated with each cylinder group records the space availability at the fragment level; to determine block availability, aligned fragments are examined. Figure 1 shows a piece of a map from a 4096/1024 file system.

Bits in map	XXXX	XXOO	OOXX	OOOO
Fragment numbers	0-3	4-7	8-11	12-15
Block numbers	0	1	2	3

Figure 1 – Example layout of blocks and fragments in a 4096/1024 file system.

Each bit in the map records the status of a fragment; an "X" shows that the fragment is in use, while a "O" shows that the fragment is available for allocation. In this example, fragments 0–5, 10, and 11 are in use, while fragments 6–9, and 12–15 are free. Fragments of adjoining blocks cannot be used as a block, even if they are large enough. In this example, fragments 6–9 cannot be coalesced into a block; only fragments 12–15 are available for allocation as a block.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remainder of the block is made available for allocation to other files. As an example consider an 11000 byte file stored on a 4096/1024 byte file system. This file would uses two full size blocks and a 3072 byte fragment. If no 3072 byte fragments are available at the time the file is created, a full size block is split yielding the necessary 3072 byte fragment and an unused 1024 byte fragment. This remaining fragment can be allocated to another file as needed.

The granularity of allocation is the *write* system call. Each time data is written to a file, the system checks to see if the size of the file has increased*. If the file needs to hold the new data, one of three conditions exists:

1) There is enough space left in an already allocated block to hold the new data. The new data is written into the available space in the block.

2) Nothing has been allocated. If the new data contains more than 4096 bytes, a 4096 byte block is allocated and the first 4096 bytes of new data is written there. This process is repeated until less than 4096 bytes of new data remain. If the remaining new data to be written will fit in three or fewer 1024 byte pieces, an unallocated fragment is located, otherwise a 4096 byte block is located. The new data is written into the located piece.

3) A fragment has been allocated. If the number of bytes in the new data plus the number of bytes already in the fragment exceeds 4096 bytes, a 4096 byte block is allocated. The contents of the fragment is copied to the beginning of the block and the remainder of the block is filled with the new data. The process then continues as in (2) above. If the number of bytes in the new data plus the number of bytes already in the fragment will fit in three or fewer 1024 byte pieces, an unallocated fragment is located, otherwise a 4096 byte block is located. The contents of the previous fragment appended with the new data is written into the allocated piece.

* A program may be overwriting data in the middle of an existing file in which case space will already be allocated.

The problem with allowing only a single fragment on a 4096/1024 byte file system is that data may be potentially copied up to three times as its requirements grow from a 1024 byte fragment to a 2048 byte fragment, then a 3072 byte fragment, and finally a 4096 byte block. The fragment reallocation can be avoided if the user program writes a full block at a time, except for a partial block at the end of the file. Because file systems with different block sizes may coexist on the same system, the file system interface been extended to provide the ability to determine the optimal size for a read or write. For files the optimal size is the block size of the file system on which the file is being accessed. For other objects, such as pipes and sockets, the optimal size is the underlying buffer size. This feature is used by the Standard Input/Output Library, a package used by most user programs. This feature is also used by certain system utilities such as archivers and loaders that do their own input and output management and need the highest possible file system bandwidth.

The space overhead in the 4096/1024 byte new file system organisation is empirically observed to be about the same as in the 1024 byte old file system organisation. A file system with 4096 byte blocks and 512 byte fragments has about the same amount of space overhead as the 512 byte block UNIX file system. The new file system is more space efficient than the 512 byte or 1024 byte file systems in that it uses the same amount of space for small files while requiring less indexing information for large files. This savings is offset by the need to use more space for keeping track of available free blocks. The net result is about the same disk utilisation when the new file systems fragment size equals the old file systems block size.

In order for the layout policies to be effective, the disk cannot be kept completely full. Each file system maintains a parameter that gives the minimum acceptable percentage of file system blocks that can be free. If the the number of free blocks drops below this level only the system administrator can continue to allocate blocks. The value of this parameter can be changed at any time, even when the file system is mounted and active. The transfer rates to be given in section 4 were measured on file systems kept less than 90% full. If the reserve of free blocks is set to zero, the file system throughput rate tends to be cut in half, because of the inability of the file system to localize the blocks in a file. If the performance is impaired because of overfilling, it may be restored by removing enough files to obtain 10% free space. Access speed for files created during periods of little free space can be restored by recreating them once enough space is available. The amount of free space maintained must be added to the percentage of waste when comparing the organizations given in Table 1. Thus, a site running the old 1024 byte UNIX file system wastes 11.8% of the space and one could expect to fit the same amount of data into a 4096/512 byte new file system with 5% free space, since a 512 byte old file system wasted 6.9% of the space.

3.2. File system parameterization

Except for the initial creation of the free list, the old file system does not attempt to parameterize itself to the underlying hardware. It has no information about either the physical characteristics of the mass storage device, or the hardware that interacts with it. A goal of the new file system is to parameterize the processor capabilities and mass storage characteristics so that blocks can be allocated in an optimum configuration dependent way. Parameters used include the speed of the processor, the hardware support for mass storage transfers, and the characteristics of the mass storage devices. Disk technology is constantly improving and a given installation can have several different disk technologies running on a single processor. Each file system is parameterized so that it can adapt to the characteristics of the disk on which it is placed.

For mass storage devices such as disks, the new file system tries to allocate new blocks on the same cylinder as the previous block in the same file. Optimally, these new blocks will also be well positioned rotationally. The distance between "rotationally optimal" blocks varies greatly; it can be a consecutive block or a rotationally delayed block depending on system characteristics. On a processor with a channel that does not require any processor intervention between mass storage transfer requests, two consecutive disk blocks often can be accessed without suffering lost time because of an intervening disk revolution. For processors without such channels, the main processor must field an interrupt and prepare for a new disk transfer. The expected time to

service this interrupt and schedule a new disk transfer depends on the speed of the main processor.

The physical characteristics of each disk include the number of blocks per track and the rate at which the disk spins. The allocation policy routines use this information to calculate the number of milliseconds required to skip over a block. The characteristics of the processor include the expected time to schedule an interrupt. Given the previous block allocated to a file, the allocation routines calculate the number of blocks to skip over so that the next block in a file will be coming into position under the disk head in the expected amount of time that it takes to start a new disk transfer operation. For programs that sequentially access large amounts of data, this strategy minimizes the amount of time spent waiting for the disk to position itself.

To ease the calculation of finding rotationally optimal blocks, the cylinder group summary information includes a count of the availability of blocks at different rotational positions. Eight rotational positions are distinguished, so the resolution of the summary information is 2 milliseconds for a typical 3600 revolution per minute drive.

The parameter that defines the minimum number of milliseconds between the completion of a data transfer and the initiation of another data transfer on the same cylinder can be changed at any time, even when the file system is mounted and active. If a file system is parameterized to lay out blocks with rotational separation of 2 milliseconds, and the disk pack is then moved to a system that has a processor requiring 4 milliseconds to schedule a disk operation, the throughput will drop precipitously because of lost disk revolutions on nearly every block. If the eventual target machine is known, the file system can be parameterized for it even though it is initially created on a different processor. Even if the move is not known in advance, the rotational layout delay can be reconfigured after the disk is moved so that all further allocation is done based on the characteristics of the new host.

3.3. Layout policies

The file system policies are divided into two distinct parts. At the top level are global policies that use file system wide summary information to make decisions regarding the placement of new inodes and data blocks. These routines are responsible for deciding the placement of new directories and files. They also calculate rotationally optimal block layouts, and decide when to force a long seek to a new cylinder group because there are insufficient blocks left in the current cylinder group to do reasonable layouts. Below the global policy routines are the local allocation routines that use a locally optimal scheme to lay out data blocks.

Two methods for improving file system performance are to increase the locality of reference to minimize seek latency as described by [Trivedi80], and to improve the layout of data to make larger transfers possible as described by [Nevalainen77]. The global layout policies try to improve performance by clustering related information. They cannot attempt to localize all data references, but must also try to spread unrelated data among different cylinder groups. If too much localization is attempted, the local cylinder group may run out of space forcing the data to be scattered to non-local cylinder groups. Taken to an extreme, total localization can result in a single huge cluster of data resembling the old file system. The global policies try to balance the two conflicting goals of localizing data that is concurrently accessed while spreading out unrelated data.

One allocatable resource is inodes. Inodes are used to describe both files and directories. Files in a directory are frequently accessed together. For example the "list directory" command often accesses the inode for each file in a directory. The layout policy tries to place all the files in a directory in the same cylinder group. To ensure that files are allocated throughout the disk, a different policy is used for directory allocation. A new directory is placed in the cylinder group that has a greater than average number of free inodes, and the fewest number of directories in it already. The intent of this policy is to allow the file clustering policy to succeed most of the time. The allocation of inodes within a cylinder group is done using a next free strategy. Although this allocates the inodes randomly within a cylinder group, all the inodes for each cylinder group can be read with 4 to 8 disk transfers. This puts a small and constant upper bound on the number of

disk transfers required to access all the inodes for all the files in a directory as compared to the old file system where typically, one disk transfer is needed to get the inode for each file in a directory.

The other major resource is the data blocks. Since data blocks for a file are typically accessed together, the policy routines try to place all the data blocks for a file in the same cylinder group, preferably rotationally optimally on the same cylinder. The problem with allocating all the data blocks in the same cylinder group is that large files will quickly use up available space in the cylinder group, forcing a spill over to other areas. Using up all the space in a cylinder group has the added drawback that future allocations for any file in the cylinder group will also spill to other areas. Ideally none of the cylinder groups should ever become completely full. The solution devised is to redirect block allocation to a newly chosen cylinder group when a file exceeds 32 kilobytes, and at every megabyte thereafter. The newly chosen cylinder group is selected from those cylinder groups that have a greater than average number of free blocks left. Although big files tend to be spread out over the disk, a megabyte of data is typically accessible before a long seek must be performed, and the cost of one long seek per megabyte is small.

The global policy routines call local allocation routines with requests for specific blocks. The local allocation routines will always allocate the requested block if it is free. If the requested block is not available, the allocator allocates a free block of the requested size that is rotationally closest to the requested block. If the global layout policies had complete information, they could always request unused blocks and the allocation routines would be reduced to simple bookkeeping. However, maintaining complete information is costly; thus the implementation of the global layout policy uses heuristic guesses based on partial information.

If a requested block is not available the local allocator uses a four level allocation strategy:

1) Use the available block rotationally closest to the requested block on the same cylinder.
2) If there are no blocks available on the same cylinder, use a block within the same cylinder group.
3) If the cylinder group is entirely full, quadratically rehash among the cylinder groups looking for a free block.
4) Finally if the rehash fails, apply an exhaustive search.

The use of quadratic rehash is prompted by studies of symbol table strategies used in programming languages. File systems that are parameterized to maintain at least 10% free space almost never use this strategy; file systems that are run without maintaining any free space typically have so few free blocks that almost any allocation is random. Consequently the most important characteristic of the strategy used when the file system is low on space is that it be fast.

4. Performance

Ultimately, the proof of the effectiveness of the algorithms described in the previous section is the long term performance of the new file system.

Our empiric studies have shown that the inode layout policy has been effective. When running the "list directory" command on a large directory that itself contains many directories, the number of disk accesses for inodes is cut by a factor of two. The improvements are even more dramatic for large directories containing only files, disk accesses for inodes being cut by a factor of eight. This is most encouraging for programs such as spooling daemons that access many small files, since these programs tend to flood the disk request queue on the old file system.

Table 2 summarizes the measured throughput of the new file system. Several comments need to be made about the conditions under which these tests were run. The test programs measure the rate that user programs can transfer data to or from a file without performing any processing on it. These programs must write enough data to insure that buffering in the operating system does not affect the results. They should also be run at least three times in succession; the

first to get the system into a known state and the second two to insure that the experiment has stabilized and is repeatable. The methodology and test results are discussed in detail in [Kridle83]†. The systems were running multi-user but were otherwise quiescent. There was no contention for either the cpu or the disk arm. The only difference between the UNIBUS and MASSBUS tests was the controller. All tests used an Ampex Capricorn 330 Megabyte Winchester disk. As Table 2 shows, all file system test runs were on a VAX 11/750. All file systems had been in production use for at least a month before being measured.

Type of File System	Processor and Bus Measured	Read Speed	Read Bandwidth	Read % CPU
old 1024	750/UNIBUS	29 Kbytes/sec	29/1100 3%	11%
new 4096/1024	750/UNIBUS	221 Kbytes/sec	221/1100 20%	43%
new 8192/1024	750/UNIBUS	233 Kbytes/sec	233/1100 21%	29%
new 4096/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	73%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	54%

Table 2a – Reading rates of the old and new UNIX file systems.

Type of File System	Processor and Bus Measured	Write Speed	Write Bandwidth	Write % CPU
old 1024	750/UNIBUS	48 Kbytes/sec	48/1100 4%	29%
new 4096/1024	750/UNIBUS	142 Kbytes/sec	142/1100 13%	43%
new 8192/1024	750/UNIBUS	215 Kbytes/sec	215/1100 19%	46%
new 4096/1024	750/MASSBUS	323 Kbytes/sec	323/1200 27%	94%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	95%

Table 2b – Writing rates of the old and new UNIX file systems.

Unlike the old file system, the transfer rates for the new file system do not appear to change over time. The throughput rate is tied much more strongly to the amount of free space that is maintained. The measurements in Table 2 were based on a file system run with 10% free space. Synthetic work loads suggest the performance deteriorates to about half the throughput rates given in Table 2 when no free space is maintained.

The percentage of bandwidth given in Table 2 is a measure of the effective utilization of the disk by the file system. An upper bound on the transfer rate from the disk is measured by doing 65536* byte reads from contiguous tracks on the disk. The bandwidth is calculated by comparing the data rates the file system is able to achieve as a percentage of this rate. Using this metric, the old file system is only able to use about 3-4% of the disk bandwidth, while the new file system uses up to 39% of the bandwidth.

In the new file system, the reading rate is always at least as fast as the writing rate. This is to be expected since the kernel must do more work when allocating blocks than when simply reading them. Note that the write rates are about the same as the read rates in the 8192 byte block file system; the write rates are slower than the read rates in the 4096 byte block file system. The slower write rates occur because the kernel has to do twice as many disk allocations per second, and the processor is unable to keep up with the disk transfer rate.

In contrast the old file system is about 50% faster at writing files than reading them. This is because the *write* system call is asynchronous and the kernel can generate disk transfer requests much faster than they can be serviced, hence disk transfers build up in the disk buffer cache. Because the disk buffer cache is sorted by minimum seek order, the average seek between the

† A UNIX command that is similar to the reading test that we used is, "cp file /dev/null", where "file" is eight Megabytes long.

* This number, 65536, is the maximal I/O size supported by the VAX hardware; it is a remnant of the system's PDP-11 ancestry.

scheduled disk writes is much less than they would be if the data blocks are written out in the order in which they are generated. However when the file is read, the *read* system call is processed synchronously so the disk blocks must be retrieved from the disk in the order in which they are allocated. This forces the disk scheduler to do long seeks resulting in a lower throughput rate.

The performance of the new file system is currently limited by a memory to memory copy operation because it transfers data from the disk into buffers in the kernel address space and then spends 40% of the processor cycles copying these buffers to user address space. If the buffers in both address spaces are properly aligned, this transfer can be affected without copying by using the VAX virtual memory management hardware. This is especially desirable when large amounts of data are to be transferred. We did not implement this because it would change the semantics of the file system in two major ways; user programs would be required to allocate buffers on page boundaries, and data would disappear from buffers after being written.

Greater disk throughput could be achieved by rewriting the disk drivers to chain together kernel buffers. This would allow files to be allocated to contiguous disk blocks that could be read in a single disk transaction. Most disks contain either 32 or 48 512 byte sectors per track. The inability to use contiguous disk blocks effectively limits the performance on these disks to less than fifty percent of the available bandwidth. Since each track has a multiple of sixteen sectors it holds exactly two or three 8192 byte file system blocks, or four or six 4096 byte file system blocks. If the the next block for a file cannot be laid out contiguously, then the minimum spacing to the next allocatable block on any platter is between a sixth and a half a revolution. The implication of this is that the best possible layout without contiguous blocks uses only half of the bandwidth of any given track. If each track contains an odd number of sectors, then it is possible to resolve the rotational delay to any number of sectors by finding a block that begins at the desired rotational position on another track. The reason that block chaining has not been implemented is because it would require rewriting all the disk drivers in the system, and the current throughput rates are already limited by the speed of the available processors.

Currently only one block is allocated to a file at a time. A technique used by the DEMOS file system when it finds that a file is growing rapidly, is to preallocate several blocks at once, releasing them when the file is closed if they remain unused. By batching up the allocation the system can reduce the overhead of allocating at each write, and it can cut down on the number of disk writes needed to keep the block pointers on the disk synchronized with the block allocation [Powell79].

5. File system functional enhancements

The speed enhancements to the UNIX file system did not require any changes to the semantics or data structures viewed by the users. However several changes have been generally desired for some time but have not been introduced because they would require users to dump and restore all their file systems. Since the new file system already requires that all existing file systems be dumped and restored, these functional enhancements have been introduced at this time.

5.1. Long file names

File names can now be of nearly arbitrary length. The only user programs affected by this change are those that access directories. To maintain portability among UNIX systems that are not running the new file system, a set of directory access routines have been introduced that provide a uniform interface to directories on both old and new systems.

Directories are allocated in units of 512 bytes. This size is chosen so that each allocation can be transferred to disk in a single atomic operation. Each allocation unit contains variable-length directory entries. Each entry is wholly contained in a single allocation unit. The first three fields of a directory entry are fixed and contain an inode number, the length of the entry, and the length of the name contained in the entry. Following this fixed size information is the

null terminated name, padded to a 4 byte boundary. The maximum length of a name in a directory is currently 255 characters.

Free space in a directory is held by entries that have a record length that exceeds the space required by the directory entry itself. All the bytes in a directory unit are claimed by the directory entries. This normally results in the last entry in a directory being large. When entries are deleted from a directory, the space is returned to the previous entry in the same directory unit by increasing its length. If the first entry of a directory unit is free, then its inode number is set to zero to show that it is unallocated.

5.3. File locking

The old file system had no provision for locking files. Processes that needed to synchronize the updates of a file had to create a separate "lock" file to synchronise their updates. A process would try to create a "lock" file. If the creation succeeded, then it could proceed with its update; if the creation failed, then it would wait, and try again. This mechanism had three drawbacks. Processes consumed CPU time by looping over attempts to create locks. Locks were left lying around following system crashes and had to be cleaned up by hand. Finally, processes running as system administrator are always permitted to create files, so they had to use a different mechanism. While it is possible to get around all these problems, the solutions are not straight-forward, so a mechanism for locking files has been added.

The most general schemes allow processes to concurrently update a file. Several of these techniques are discussed in [Peterson83]. A simpler technique is to simply serialize access with locks. To attain reasonable efficiency, certain applications require the ability to lock pieces of a file. Locking down to the byte level has been implemented in the Onyx file system by [Bass81]. However, for the applications that currently run on the system, a mechanism that locks at the granularity of a file is sufficient.

Locking schemes fall into two classes, those using hard locks and those using advisory locks. The primary difference between advisory locks and hard locks is the decision of when to override them. A hard lock is always enforced whenever a program tries to access a file; an advisory lock is only applied when it is requested by a program. Thus advisory locks are only effective when all programs accessing a file use the locking scheme. With hard locks there must be some override policy implemented in the kernel, with advisory locks the policy is implemented by the user programs. In the UNIX system, programs with system administrator privilege can override any protection scheme. Because many of the programs that need to use locks run as system administrators, we chose to implement advisory locks rather than create a protection scheme that was contrary to the UNIX philosophy or could not be used by system administration programs.

The file locking facilities allow cooperating programs to apply advisory *shared* or *exclusive* locks on files. Only one process has an exclusive lock on a file while multiple shared locks may be present. Both shared and exclusive locks cannot be present on a file at the same time. If any lock is requested when another process holds an exclusive lock, or an exclusive lock is requested when another process holds any lock, the open will block until the lock can be gained. Because shared and exclusive locks are advisory only, even if a process has obtained a lock on a file, another process can override the lock by opening the same file without a lock.

Locks can be applied or removed on open files, so that locks can be manipulated without needing to close and reopen the file. This is useful, for example, when a process wishes to open a file with a shared lock to read some information, to determine whether an update is required. It can then get an exclusive lock so that it can do a read, modify, and write to update the file in a consistent manner.

A request for a lock will cause the process to block if the lock can not be immediately obtained. In certain instances this is unsatisfactory. For example, a process that wants only to check if a lock is present would require a separate mechanism to find out this information. Consequently, a process may specify that its locking request should return with an error if a lock can not be immediately obtained. Being able to poll for a lock is useful to "daemon" processes that

wish to service a spooling area. If the first instance of the daemon locks the directory where spooling takes place, later daemon processes can easily check to see if an active daemon exists. Since the lock is removed when the process exits or the system crashes, there is no problem with unintentional locks files that must be cleared by hand.

Almost no deadlock detection is attempted. The only deadlock detection made by the system is that the file descriptor to which a lock is applied does not currently have a lock of the same type (i.e. the second of two successive calls to apply a lock of the same type will fail). Thus a process can deadlock itself by requesting locks on two separate file descriptors for the same object.

5.3. Symbolic links

The 512 byte UNIX file system allows multiple directory entries in the same file system to reference a single file. The link concept is fundamental; files do not live in directories, but exist separately and are referenced by links. When all the links are removed, the file is deallocated. This style of links does not allow references across physical file systems, nor does it support inter-machine linkage. To avoid these limitations *symbolic links* have been added similar to the scheme used by Multics [Feiertag71].

A symbolic link is implemented as a file that contains a pathname. When the system encounters a symbolic link while interpreting a component of a pathname, the contents of the symbolic link is prepended to the rest of the pathname, and this name is interpreted to yield the resulting pathname. If the symbolic link contains an absolute pathname, the absolute pathname is used, otherwise the contents of the symbolic link is evaluated relative to the location of the link in the file hierarchy.

Normally programs do not want to be aware that there is a symbolic link in a pathname that they are using. However certain system utilities must be able to detect and manipulate symbolic links. Three new system calls provide the ability to detect, read, and write symbolic links, and seven system utilities were modified to use these calls.

In future Berkeley software distributions it will be possible to mount file systems from other machines within a local file system. When this occurs, it will be possible to create symbolic links that span machines.

5.4. Rename

Programs that create new versions of data files typically create the new version as a temporary file and then rename the temporary file with the original name of the data file. In the old UNIX file systems the renaming required three calls to the system. If the program were interrupted or the system crashed between these calls, the data file could be left with only its temporary name. To eliminate this possibility a single system call has been added that performs the rename in an atomic fashion to guarantee the existence of the original name.

In addition, the rename facility allows directories to be moved around in the directory tree hierarchy. The rename system call performs special validation checks to insure that the directory tree structure is not corrupted by the creation of loops or inaccessible directories. Such corruption would occur if a parent directory were moved into one of its descendants. The validation check requires tracing the ancestry of the target directory to insure that it does not include the directory being moved.

5.5. Quotas

The UNIX system has traditionally attempted to share all available resources to the greatest extent possible. Thus any single user can allocate all the available space in the file system. In certain environments this is unacceptable. Consequently, a quota mechanism has been added for restricting the amount of file system resources that a user can obtain. The quota mechanism sets limits on both the number of files and the number of disk blocks that a user may allocate. A separate quota can be set for each user on each file system. Each resource is given both a hard

and a soft limit. When a program exceeds a soft limit, a warning is printed on the users terminal; the offending program is not terminated unless it exceeds its hard limit. The idea is that users should stay below their soft limit between login sessions, but they may use more space while they are actively working. To encourage this behavior, users are warned when logging in if they are over any of their soft limits. If they fail to correct the problem for too many login sessions, they are eventually reprimanded by having their soft limit enforced as their hard limit.

Appendix C – Networking Implementation

1. Introduction

This report describes the internal structure of facilities added to the 4.2BSD version of the UNIX operating system for the VAX. The system facilities provide a uniform user interface to networking within UNIX. In addition, the implementation introduces a structure for network communications which may be used by system implementors in adding new networking facilities. The internal structure is not visible to the user, rather it is intended to aid implementors of communication protocols and network services by providing a framework which promotes code sharing and minimises implementation effort.

The reader is expected to be familiar with the C programming language and system interface, as described in the body of this report. Basic understanding of network communication concepts is assumed; where required any additional ideas are introduced.

The remainder of this document provides a description of the system internals, avoiding, when possible, those portions which are utilised only by the interprocess communication facilities.

2. Overview

If we consider the International Standards Organisation's (ISO) Open System Interconnection (OSI) model of network communication [ISO81] [Zimmermann80], the networking facilities described here correspond to a portion of the session layer (layer 3) and all of the transport and network layers (layers 2 and 1, respectively).

The network layer provides possibly imperfect data transport services with minimal addressing structure. Addressing at this level is normally host to host, with implicit or explicit routing optionally supported by the communicating agents.

At the transport layer the notions of reliable transfer, data sequencing, flow control, and service addressing are normally included. Reliability is usually managed by explicit acknowledgement of data delivered. Failure to acknowledge a transfer results in retransmission of the data. Sequencing may be handled by tagging each message handed to the network layer by a *sequence number* and maintaining state at the endpoints of communication to utilize received sequence numbers in reordering data which arrives out of order.

The session layer facilities may provide forms of addressing which are mapped into formats required by the transport layer, service authentication and client authentication, etc. Various systems also provide services such as data encryption and address and protocol translation.

The following sections begin by describing some of the common data structures and utility routines, then examine the internal layering. The contents of each layer and its interface are considered. Certain of the interfaces are protocol implementation specific. For these cases examples have been drawn from the Internet [Cerf78] protocol family. Later sections cover routing issues, the design of the raw socket interface and other miscellaneous topics.

3. Goals

The networking system was designed with the goal of supporting multiple *protocol families* and addressing styles. This required information to be "hidden" in common data structures which could be manipulated by all the pieces of the system, but which required interpretation only by the protocols which "controlled" it. The system described here attempts to minimise the use of shared data structures to those kept by a suite of protocols (a *protocol family*), and those used for rendesvous between "synchronous" and "asynchronous" portions of the system (e.g. queues of data packets are filled at interrupt time and emptied based on user requests).

A major goal of the system was to provide a framework within which new protocols and hardware could be easily be supported. To this end, a great deal of effort has been extended to create utility routines which hide many of the more complex and/or hardware dependent chores of networking. Later sections describe the utility routines and the underlying data structures they manipulate.

4. Internal address representation

Common to all portions of the system are two data structures. These structures are used to represent addresses and various data objects. Addresses, internally are described by the *sockaddr* structure,

```
struct sockaddr {
        short sa_family;  /* data format identifier */
        char  sa_data[14];      /* address */
};
```

All addresses belong to one or more *address families* which define their format and interpretation. The *sa_family* field indicates which address family the address belongs to, the *sa_data* field contains the actual data value. The size of the data field, 14 bytes, was selected based on a study of current address formats*.

5. Memory management

A single mechanism is used for data storage: memory buffers, or *mbufs*. An mbuf is a structure of the form:

```
struct mbuf {
        struct      mbuf *m_next;       /* next buffer in chain */
        u_long      m_off;              /* offset of data */
        short m_len;                  /* amount of data in this mbuf */
        short m_type;                 /* mbuf type (accounting) */
        u_char      m_dat[MLEN];        /* data storage */
        struct      mbuf *m_act;        /* link in higher-level mbuf list */
};
```

The *m_next* field is used to chain mbufs together on linked lists, while the *m_act* field allows lists of mbufs to be accumulated. By convention, the mbufs common to a single object (for example, a packet) are chained together with the *m_next* field, while groups of objects are linked via the *m_act* field (possibly when in a queue).

Each mbuf has a small data area for storing information, *m_dat*. The *m_len* field indicates the amount of data, while the *m_off* field is an offset to the beginning of the data from the base of the mbuf. Thus, for example, the macro *mtod*, which converts a pointer to an mbuf to a pointer to the data stored in the mbuf, has the form

```
#define     mtod(x,t)   ((t)((int)(x) + (x)->m_off))
```

(note the *t* parameter, a C type cast, is used to cast the resultant pointer for proper assignment).

In addition to storing data directly in the mbuf's data area, data of page size may be also be stored in a separate area of memory. The mbuf utility routines maintain a pool of pages for this purpose and manipulate a private page map for such pages. The virtual addresses of these data pages precede those of mbufs, so when pages of data are separated from an mbuf, the mbuf data

* Later versions of the system support variable length addresses.

offset is a negative value. An array of reference counts on pages is also maintained so that copies of pages may be made without core to core copying (copies are created simply by duplicating the relevant page table entries in the data page map and incrementing the associated reference counts for the pages). Separate data pages are currently used only when copying data from a user process into the kernel, and when bringing data in at the hardware level. Routines which manipulate mbufs are not normally aware if data is stored directly in the mbuf data array, or if it is kept in separate pages.

The following utility routines are available for manipulating mbuf chains:

m = m_copy(m0, off, len);

The *m_copy* routine create a copy of all, or part, of a list of the mbufs in *m0*. *Len* bytes of data, starting *off* bytes from the front of the chain, are copied. Where possible, reference counts on pages are used instead of core to core copies. The original mbuf chain must have at least *off* + *len* bytes of data. If *len* is specified as M_COPYALL, all the data present, offset as before, is copied.

m_cat(m, n);

The mbuf chain, *n*, is appended to the end of *m*. Where possible, compaction is performed.

m_adj(m, diff);

The mbuf chain, *m* is adjusted in size by *diff* bytes. If *diff* is non-negative, *diff* bytes are shaved off the front of the mbuf chain. If *diff* is negative, the alteration is performed from back to front. No space is reclaimed in this operation, alterations are accomplished by changing the *m_len* and *m_off* fields of mbufs.

m = m_pullup(m0, size);

After a successful call to *m_pullup*, the mbuf at the head of the returned list, *m*, is guaranteed to have at least *size* bytes of data in contiguous memory (allowing access via a pointer, obtained using the *mtod* macro). If the original data was less than *size* bytes long, *len* was greater than the size of an mbuf data area (112 bytes), or required resources were unavailable, *m* is 0 and the original mbuf chain is deallocated.

This routine is particularly useful when verifying packet header lengths on reception. For example, if a packet is received and only 8 of the necessary 16 bytes required for a valid packet header are present at the head of the list of mbufs representing the packet, the remaining 8 bytes may be "pulled up" with a single *m_pullup* call. If the call fails the invalid packet will have been discarded.

By insuring mbufs always reside on 128 byte boundaries it is possible to always locate the mbuf associated with a data area by masking off the low bits of the virtual address. This allows modules to store data structures in mbufs and pass them around without concern for locating the original mbuf when it comes time to free the structure. The *dtom* macro is used to convert a pointer into an mbuf's data area to a pointer to the mbuf,

```
#define    dtom(x)    ((struct mbuf *)((int)x & ~(MSIZE-1)))
```

Mbufs are used for dynamically allocated data structures such as sockets, as well as memory allocated for packets. Statistics are maintained on mbuf usage and can be viewed by users using the *netstat*(1) program.

8. Internal layering

The internal structure of the network system is divided into three layers. These layers correspond to the services provided by the socket abstraction, those provided by the communication protocols, and those provided by the hardware interfaces. The communication protocols are normally layered into two or more individual cooperating layers, though they are collectively viewed in the system as one layer providing services supportive of the appropriate socket abstraction.

The following sections describe the properties of each layer in the system and the interfaces each must conform to.

8.1. Socket layer

The socket layer deals with the interprocess communications facilities provided by the system. A socket is a bidirectional endpoint of communication which is "typed" by the semantics of communication it supports. The system calls described in the *4.2BSD System Manual* are used to manipulate sockets.

A socket consists of the following data structure:

```
struct socket {
        short so_type;          /* generic type */
        short so_options;       /* from socket call */
        short so_linger;        /* time to linger while closing */
        short so_state;         /* internal state flags */
        caddr_t     so_pcb;             /* protocol control block */
        struct      protosw *so_proto;  /* protocol handle */
        struct      socket *so_head; /* back pointer to accept socket */
        struct      socket *so_q0;      /* queue of partial connections */
        short so_q0len;         /* partials on so_q0 */
        struct      socket *so_q;       /* queue of incoming connections */
        short so_qlen;          /* number of connections on so_q */
        short so_qlimit;        /* max number queued connections */
        struct      sockbuf so_snd;     /* send queue */
        struct      sockbuf so_rcv;     /* receive queue */
        short so_timeo;         /* connection timeout */
        u_short     so_error;       /* error affecting connection */
        short so_oobmark;           /* chars to oob mark */
        short so_pgrp;          /* pgrp for signals */
};
```

Each socket contains two data queues, *so_rcv* and *so_snd*, and a pointer to routines which provide supporting services. The type of the socket, *so_type* is defined at socket creation time and used in selecting those services which are appropriate to support it. The supporting protocol is selected at socket creation time and recorded in the socket data structure for later use. Protocols are defined by a table of procedures, the *protosw* structure, which will be described in detail later. A pointer to a protocol specific data structure, the "protocol control block" is also present in the socket structure. Protocols control this data structure and it normally includes a back pointer to the parent socket structure(s) to allow easy lookup when returning information to a user (for example, placing an error number in the *so_error* field). The other entries in the socket structure are used in queueing connection requests, validating user requests, storing socket characteristics (e.g. options supplied at the time a socket is created), and maintaining a socket's state.

Processes "rendezvous at a socket" in many instances. For instance, when a process wishes to extract data from a socket's receive queue and it is empty, or lacks sufficient data to satisfy the request, the process blocks, supplying the address of the receive queue as an "wait channel' to be used in notification. When data arrives for the process and is placed in the socket's queue, the blocked process is identified by the fact it is waiting "on the queue".

8.1.1. Socket state

A socket's state is defined from the following:

```
#define  SS_NOFDREF            0x001   /* no file table ref any more */
#define  SS_ISCONNECTED        0x002   /* socket connected to a peer */
#define  SS_ISCONNECTING       0x004   /* in process of connecting to peer */
#define  SS_ISDISCONNECTING    0x008   /* in process of disconnecting */
#define  SS_CANTSENDMORE       0x010   /* can't send more data to peer */
#define  SS_CANTRCVMORE        0x020   /* can't receive more data from peer */
#define  SS_CONNAWAITING       0x040   /* connections awaiting acceptance */
#define  SS_RCVATMARK          0x080   /* at mark on input */

#define  SS_PRIV               0x100   /* privileged */
#define  SS_NBIO               0x200   /* non-blocking ops */
#define  SS_ASYNC              0x400   /* async i/o notify */
```

The state of a socket is manipulated both by the protocols and the user (through system calls). When a socket is created the state is defined based on the type of input/output the user wishes to perform. "Non-blocking" I/O implies a process should never be blocked to await resources. Instead, any call which would block returns prematurely with the error EWOULDBLOCK (the service request may be partially fulfilled, e.g. a request for more data than is present).

If a process requested "asynchronous" notification of events related to the socket the SIGIO signal is posted to the process. An event is a change in the socket's state, examples of such occurrances are: space becoming available in the send queue, new data available in the receive queue, connection establishment or disestablishment, etc.

A socket may be marked "priviledged" if it was created by the super-user. Only priviledged sockets may send broadcast packets, or bind addresses in priviledged portions of an address space.

6.1.2. Socket data queues

A socket's data queue contains a pointer to the data stored in the queue and other entries related to the management of the data. The following structure defines a data queue:

```
struct sockbuf {
        short sb_cc;            /* actual chars in buffer */
        short sb_hiwat;  /* max actual char count */
        short sb_mbcnt;  /* chars of mbufs used */
        short sb_mbmax;/* max chars of mbufs to use */
        short sb_lowat;  /* low water mark */
        short sb_timeo;  /* timeout */
        struct      mbuf *sb_mb;    /* the mbuf chain */
        struct      proc *sb_sel;   /* process selecting read/write */
        short sb_flags;  /* flags, see below */
};
```

Data is stored in a queue as a chain of mbufs. The actual count of characters as well as high and low water marks are used by the protocols in controlling the flow of data. The socket routines cooperate in implementing the flow control policy by blocking a process when it requests to send data and the high water mark has been reached, or when it requests to receive data and less than the low water mark is present (assuming non-blocking I/O has not been specified).

When a socket is created, the supporting protocol "reserves" space for the send and receive queues of the socket. The actual storage associated with a socket queue may fluctuate during a socket's lifetime, but is assumed this reservation will always allow a protocol to acquire enough memory to satisfy the high water marks.

The timeout and select values are manipulated by the socket routines in implementing various portions of the interprocess communications facilities and will not be described here.

A socket queue has a number of flags used in synchronizing access to the data and in acquiring resources;

```
#define   SB_LOCK 0x01  /* lock on data queue (so_rcv only) */
#define   SB_WANT0x02   /* someone is waiting to lock */
#define   SB_WAIT 0x04  /* someone is waiting for data/space */
#define   SB_SEL  0x08  /* buffer is selected */
#define   SB_COLL 0x10  /* collision selecting */
```

The last two flags are manipulated by the system in implementing the select mechanism.

8.1.8. Socket connection queueing

In dealing with connection oriented sockets (e.g. SOCK_STREAM) the two sides are considered distinct. One side is termed *active*, and generates connection requests. The other side is called *passive* and accepts connection requests.

From the passive side, a socket is created with the option SO_ACCEPTCONN specified, creating two queues of sockets: *so_q0* for connections in progress and *so_q* for connections already made and awaiting user acceptance. As a protocol is preparing incoming connections, it creates a socket structure queued on *so_q0* by calling the routine *sonewconn*(). When the connection is established, the socket structure is then transfered to *so_q*, making it available for an accept.

If an SO_ACCEPTCONN socket is closed with sockets on either *so_q0* or *so_q*, these sockets are dropped.

8.2. Protocol layer(s)

Protocols are described by a set of entry points and certain socket visible characteristics, some of which are used in deciding which socket type(s) they may support.

An entry in the "protocol switch" table exists for each protocol module configured into the system. It has the following form:

```
struct protosw {
        short pr_type;          /* socket type used for */
        short pr_family;        /* protocol family */
        short pr_protocol;              /* protocol number */
        short pr_flags;         /* socket visible attributes */
/* protocol-protocol hooks */
        int     (*pr_input)();          /* input to protocol (from below) */
        int     (*pr_output)();         /* output to protocol (from above) */
        int     (*pr_ctlinput)(); /* control input (from below) */
        int     (*pr_ctloutput)();      /* control output (from above) */
/* user-protocol hook */
        int     (*pr_usrreq)();         /* user request */
/* utility hooks */
        int     (*pr_init)();           /* initialization routine */
        int     (*pr_fasttimo)(); /* fast timeout (200ms) */
        int     (*pr_slowtimo)();/* slow timeout (500ms) */
        int     (*pr_drain)();          /* flush any excess space possible */
};
```

A protocol is called through the *pr_init* entry before any other. Thereafter it is called every 200 milliseconds through the *pr_fasttimo* entry and every 500 milliseconds through the *pr_slowtimo* for timer based actions. The system will call the *pr_drain* entry if it is low on space and this should throw away any non-critical data.

Protocols pass data between themselves as chains of mbufs using the *pr_input* and *pr_output* routines. *Pr_input* passes data up (towards the user) and *pr_output* passes it down (towards the

network); control information passes up and down on *pr_ctlinput* and *pr_ctloutput*. The protocol is responsible for the space occupied by any the arguments to these entries and must dispose of it.

The *pr_usrreq* routine interfaces protocols to the socket code and is described below.

The *pr_flags* field is constructed from the following values:

```
#define PR_ATOMIC       0x01    /* exchange atomic messages only */
#define PR_ADDR 0x02    /* addresses given with messages */
#define PR_CONNREQUIRED     0x04    /* connection required by protocol */
#define PR_WANTRCVD     0x08    /* want PRU_RCVD calls */
#define PR_RIGHTS       0x10    /* passes capabilities */
```

Protocols which are connection-based specify the PR_CONNREQUIRED flag so that the socket routines will never attempt to send data before a connection has been established. If the PR_WANTRCVD flag is set, the socket routines will notfiy the protocol when the user has removed data from the socket's receive queue. This allows the protocol to implement acknowledgement on user receipt, and also update windowing information based on the amount of space available in the receive queue. The PR_ADDR field indicates any data placed in the socket's receive queue will be preceded by the address of the sender. The PR_ATOMIC flag specifies each *user* request to send data must be performed in a single *protocol* send request; it is the protocol's responsibility to maintain record boundaries on data to be sent. The PR_RIGHTS flag indicates the protocol supports the passing of capabilities; this is currently used only the protocols in the UNIX protocol family.

When a socket is created, the socket routines scan the protocol table looking for an appropriate protocol to support the type of socket being created. The *pr_type* field contains one of the possible socket types (e.g. SOCK_STREAM), while the *pr_family* field indicates which protocol family the protocol belongs to. The *pr_protocol* field contains the protocol number of the protocol, normally a well known value.

8.3. Network-interface layer

Each network-interface configured into a system defines a path through which packets may be sent and received. Normally a hardware device is associated with this interface, though there is no requirement for this (for example, all systems have a software "loopback" interface used for debugging and performance analysis). In addition to manipulating the hardware device, an interface module is responsible for encapsulation and deencapsulation of any low level header information required to deliver a message to it's destination. The selection of which interface to use in delivering packets is a routing decision carried out at a higher level than the network-interface layer. Each interface normally identifies itself at boot time to the routing module so that it may be selected for packet delivery.

An interface is defined by the following structure,

```
struct ifnet {
        char    *if_name;       /* name, e.g. "en" or "lo" */
        short   if_unit;        /* sub-unit for lower level driver */
        short   if_mtu;         /* maximum transmission unit */
        int     if_net;         /* network number of interface */
        short   if_flags;       /* up/down, broadcast, etc. */
        short   if_timer;       /* time 'til if_watchdog called */
        int     if_host[2];     /* local net host number */
        struct  sockaddr if_addr;       /* address of interface */
        union {
                struct  sockaddr ifu_broadaddr;
                struct  sockaddr ifu_dstaddr;
        } if_ifu;
        struct  ifqueue if_snd;         /* output queue */
        int     (*if_init)();   /* init routine */
        int     (*if_output)();         /* output routine */
        int     (*if_ioctl)();          /* ioctl routine */
        int     (*if_reset)();          /* bus reset routine */
        int     (*if_watchdog)();/* timer routine */
        int     if_ipackets,    /* packets received on interface */
        int     if_ierrors;     /* input errors on interface */
        int     if_opackets;            /* packets sent on interface */
        int     if_oerrors;     /* output errors on interface */
        int     if_collisions;          /* collisions on csma interfaces */
        struct  ifnet *if_next;
};
```

Each interface has a send queue and routines used for initialization, *if_init*, and output, *if_output*. If the interface resides on a system bus, the routine *if_reset* will be called after a bus reset has been performed. An interface may also specify a timer routine, *if_watchdog*, which should be called every *if_timer* seconds (if non-zero).

The state of an interface and certain characteristics are stored in the *if_flags* field. The following values are possible:

```
#define IFF_UP          0x1     /* interface is up */
#define IFF_BROADCAST   0x2     /* broadcast address valid */
#define IFF_DEBUG       0x4     /* turn on debugging */
#define IFF_ROUTE       0x8     /* routing entry installed */
#define IFF_POINTOPOINT 0x10    /* interface is point-to-point link */
#define IFF_NOTRAILERS  0x20    /* avoid use of trailers */
#define IFF_RUNNING     0x40    /* resources allocated */
#define IFF_NOARP       0x80    /* no address resolution protocol */
```

If the interface is connected to a network which supports transmission of *broadcast* packets, the IFF_BROADCAST flag will be set and the *if_broadaddr* field will contain the address to be used in sending or accepting a broadcast packet. If the interface is associated with a point to point hardware link (for example, a DEC DMR-11), the IFF_POINTOPOINT flag will be set and *if_dstaddr* will contain the address of the host on the other side of the connection. These addresses and the local address of the interface, *if_addr*, are used in filtering incoming packets. The interface sets IFF_RUNNING after it has allocated system resources and posted an initial read on the device it manages. This state bit is used to avoid multiple allocation requests when an interface's address is changed. The IFF_NOTRAILERS flag indicates the interface should refrain from using a *trailer* encapsulation on outgoing packets; *trailer* protocols are described in section 14. The IFF_NOARP flag indicates the interface should not use an "address resolution protocol" in mapping internetwork addresses to local network addresses.

The information stored in an *ifnet* structure for point to point communication devices is not currently used by the system internally. Rather, it is used by the user level routing process in determining host network connections and in initially devising routes (refer to chapter 10 for more information).

Various statistics are also stored in the interface structure. These may be viewed by users using the *netstat*(1) program.

The interface address and flags may be set with the SIOCSIFADDR and SIOCSIFFLAGS ioctls. SIOCSIFADDR is used to initially define each interface's address; SIOGSIFFLAGS can be used to mark an interface down and perform site-specific configuration.

8.3.1. UNIBUS interfaces

All hardware related interfaces currently reside on the UNIBUS. Consequently a common set of utility routines for dealing with the UNIBUS has been developed. Each UNIBUS interface utilises a structure of the following form:

```
struct  ifuba {
        short       ifu_uban;               /* uba number */
        short       ifu_hlen;               /* local net header length */
        struct      uba_regs *ifu_uba;      /* uba regs, in vm */
        struct ifrw {
                    caddr_t     ifrw_addr;  /* virt addr of header */
                    int         ifrw_bdp;   /* unibus bdp */
                    int         ifrw_info;  /* value from ubaalloc */
                    int         ifrw_proto; /* map register prototype */
                    struct      pte *ifrw_mr;/* base of map registers */
        } ifu_r, ifu_w;
        struct      pte ifu_wmap[IF_MAXNUBAMR];/* base pages for output */
        short       ifu_xswapd;             /* mask of clusters swapped */
        short       ifu_flags;              /* used during uballoc's */
        struct      mbuf *ifu_xtofree;      /* pages being dma'd out */
};
```

The *if_uba* structure describes UNIBUS resources held by an interface. IF_NUBAMR map registers are held for datagram data, starting at *ifr_mr*. UNIBUS map register *ifr_mr*[-1] maps the local network header ending on a page boundary. UNIBUS data paths are reserved for read and for write, given by *ifr_bdp*. The prototype of the map registers for read and for write is saved in *ifr_proto*.

When write transfers are not full pages on page boundaries the data is just copied into the pages mapped on the UNIBUS and the transfer is started. If a write transfer is of a (1024 byte) page size and on a page boundary, UNIBUS page table entries are swapped to reference the pages, and then the initial pages are remapped from *ifu_wmap* when the transfer completes.

When read transfers give whole pages of data to be input, page frames are allocated from a network page list and traded with the pages already containing the data, mapping the allocated pages to replace the input pages for the next UNIBUS data input.

The following utility routines are available for use in writing network interface drivers, all use the *ifuba* structure described above.

if_ubainit(ifu, uban, hlen, nmr);

if_ubainit allocates resources on UNIBUS adaptor *uban* and stores the resultant information in the *ifuba* structure pointed to by *ifu*. It is called only at boot time or after a UNIBUS reset. Two data paths (buffered or unbuffered, depending on the *ifu_flags* field) are allocated, one for reading and one for writing. The *nmr* parameter indicates the number of UNIBUS mapping registers required to map a maximal sized packet onto the UNIBUS, while *hlen* specifies the size of a local network header, if any, which should be mapped separately from

the data (see the description of trailer protocols in chapter 14). Sufficient UNIBUS mapping registers and pages of memory are allocated to initialize the input data path for an initial read. For the output data path, mapping registers and pages of memory are also allocated and mapped onto the UNIBUS. The pages associated with the output data path are held in reserve in the event a write requires copying non-page-aligned data (see *if_wubaput* below). If *if_ubainit* is called with resources already allocated, they will be used instead of allocating new ones (this normally occurs after a UNIBUS reset). A 1 is returned when allocation and initialization is successful, 0 otherwise.

m = if_rubaget(ifu, totlen, off0);

if_rubaget pulls read data off an interface. *totlen* specifies the length of data to be obtained, not counting the local network header. If *off0* is non-zero, it indicates a byte offset to a trailing local network header which should be copied into a separate mbuf and prepended to the front of the resultant mbuf chain. When page sized units of data are present and are page-aligned, the previously mapped data pages are remapped into the mbufs and swapped with fresh pages; thus avoiding any copying. A 0 return value indicates a failure to allocate resources.

if_wubaput(ifu, m);

if_wubaput maps a chain of mbufs onto a network interface in preparation for output. The chain includes any local network header, which is copied so that it resides in the mapped and aligned I/O space. Any other mbufs which contained non page sized data portions are also copied to the I/O space. Pages mapped from a previous output operation (no longer needed) are unmapped and returned to the network page pool.

7. Socket/protocol interface

The interface between the socket routines and the communication protocols is through the *pr_usrreq* routine defined in the protocol switch table. The following requests to a protocol module are possible:

```
#define PRU_ATTACH 0    /* attach protocol */
#define PRU_DETACH 1    /* detach protocol */
#define PRU_BIND    2   /* bind socket to address */
#define PRU_LISTEN  3   /* listen for connection */
#define PRU_CONNECT     4   /* establish connection to peer */
#define PRU_ACCEPT 5    /* accept connection from peer */
#define PRU_DISCONNECT 6    /* disconnect from peer */
#define PRU_SHUTDOWN    7   /* won't send any more data */
#define PRU_RCVD    8   /* have taken data; more room now */
#define PRU_SEND    9   /* send this data */
#define PRU_ABORT   10  /* abort (fast DISCONNECT, DETATCH) */
#define PRU_CONTROL     11  /* control operations on protocol */
#define PRU_SENSE   12  /* return status into m */
#define PRU_RCVOOB 13   /* retrieve out of band data */
#define PRU_SENDOOB     14  /* send out of band data */
#define PRU_SOCKADDR    15  /* fetch socket's address */
#define PRU_PEERADDR    16  /* fetch peer's address */
#define PRU_CONNECT2    17  /* connect two sockets */
/* begin for protocols internal use */
#define PRU_FASTTIMO    18  /* 200ms timeout */
#define PRU_SLOWTIMO    19  /* 500ms timeout */
#define PRU_PROTORCV    20  /* receive from below */
#define PRU_PROTOSEND   21  /* send to below */
```

A call on the user request routine is of the form,

```
error = (*protosw[].pr_usrreq)(up, req, m, addr, rights);
int error; struct socket *up; int req; struct mbuf *m, *rights; caddr_t addr;
```

The mbuf chain, *m*, and the address are optional parameters. The *rights* parameter is an optional pointer to an mbuf chain containing user specified capabilities (see the *sendmsg* and *recvmsg* system calls). The protocol is responsible for disposal of both mbuf chains. A non-zero return value gives a UNIX error number which should be passed to higher level software. The following paragraphs describe each of the requests possible.

PRU_ATTACH
: When a protocol is bound to a socket (with the *socket* system call) the protocol module is called with this request. It is the responsibility of the protocol module to allocate any resources necessary. The "attach" request will always precede any of the other requests, and should not occur more than once.

PRU_DETACH
: This is the antithesis of the attach request, and is used at the time a socket is deleted. The protocol module may deallocate any resources assigned to the socket.

PRU_BIND
: When a socket is initially created it has no address bound to it. This request indicates an address should be bound to an existing socket. The protocol module must verify the requested address is valid and available for use.

PRU_LISTEN
: The "listen" request indicates the user wishes to listen for incoming connection requests on the associated socket. The protocol module should perform any state changes needed to carry out this request (if possible). A "listen" request always precedes a request to accept a connection.

PRU_CONNECT
: The "connect" request indicates the user wants to a establish an association. The *addr* parameter supplied describes the peer to be connected to. The effect of a connect request may vary depending on the protocol. Virtual circuit protocols, such as TCP [Postel80b], use this request to initiate establishment of a TCP connection. Datagram protocols, such as UDP [Postel79], simply record the peer's address in a private data structure and use it to tag all outgoing packets. There are no restrictions on how many times a connect request may be used after an attach. If a protocol supports the notion of *multi-casting*, it is possible to use multiple connects to establish a multi-cast group. Alternatively, an association may be broken by a PRU_DISCONNECT request, and a new association created with a subsequent connect request; all without destroying and creating a new socket.

PRU_ACCEPT
: Following a successful PRU_LISTEN request and the arrival of one or more connections, this request is made to indicate the user has accepted the first connection on the queue of pending connections. The protocol module should fill in the supplied address buffer with the address of the connected party.

PRU_DISCONNECT
: Eliminate an association created with a PRU_CONNECT request.

PRU_SHUTDOWN
: This call is used to indicate no more data will be sent and/or received (the *addr* parameter indicates the direction of the shutdown, as encoded in the *soshutdown* system call). The protocol may, at its discretion, deallocate any data structures related to the shutdown.

PRU_RCVD
: This request is made only if the protocol entry in the protocol switch table includes the PR_WANTRCVD flag. When a user removes data from the receive queue this request will be sent to the protocol module. It may be used to trigger acknowledgements, refresh windowing information, initiate data transfer, etc.

PRU_SEND

Each user request to send data is tra lated into one or more PRU_SEND requests (a protocol may indicate a single user send quest must be translated into a single PRU_SEND request by specifying the PR_ATOMIC flag in its protocol description). The data to be sent is presented to the protocol as a list of mbufs and an address is, optionally, supplied in the *addr* parameter. The protocol is responsible for preserving the data in the socket's send queue if it is not able to send it immediately, or if it may need it at some later time (e.g. for retransmission).

PRU_ABORT

This request indicates an abnormal termination of service. The protocol should delete any existing association(s).

PRU_CONTROL

The "control" request is generated when a user performs a UNIX *ioctl* system call on a socket (and the ioctl is not intercepted by the socket routines). It allows protocol-specific operations to be provided outside the scope of the common socket interface. The *addr* parameter contains a pointer to a static kernel data area where relevant information may be obtained or returned. The *m* parameter contains the actual *ioctl* request code (note the non-standard calling convention).

PRU_SENSE

The "sense" request is generated when the user makes an *fstat* system call on a socket; it requests status of the associated socket. There currently is no common format for the status returned. Information which might be returned includes per-connection statistics, protocol state, resources currently in use by the connection, the optimal transfer size for the connection (based on windowing information and maximum packet size). The *addr* parameter contains a pointer to a static kernel data area where the status buffer should be placed.

PRU_RCVOOB

Any "out-of-band" data presently available is to be returned. An mbuf is passed in to the protocol module and the protocol should either place data in the mbuf or attach new mbufs to the one supplied if there is insufficient space in the single mbuf.

PRU_SENDOOB

Like PRU_SEND, but for out-of-band data.

PRU_SOCKADDR

The local address of the socket is returned, if any is currently bound to the it. The address format (protocol specific) is returned in the *addr* parameter.

PRU_PEERADDR

The address of the peer to which the socket is connected is returned. The socket must be in a SS_ISCONNECTED state for this request to be made to the protocol. The address format (protocol specific) is returned in the *addr* parameter.

PRU_CONNECT2

The protocol module is supplied two sockets and requested to establish a connection between the two without binding any addresses, if possible. This call is used in implementing the system call.

The following requests are used internally by the protocol modules and are never generated by the socket routines. In certain instances, they are handed to the *pr_usrreq* routine solely for convenience in tracing a protocol's operation (e.g. PRU_SLOWTIMO).

PRU_FASTTIMO

A "fast timeout" has occured. This request is made when a timeout occurs in the protocol's *pr_fastimo* routine. The *addr* parameter indicates which timer expired.

PRU_SLOWTIMO

A "slow timeout" has occured. This request is made when a timeout occurs in the protocol's *pr_slowtimo* routine. The *addr* parameter indicates which timer expired.

PRU_PROTORCV

This request is used in the protocol-protocol interface, not by the routines. It requests reception of data destined for the protocol and not the user. No protocols currently use this facility.

PRU_PROTOSEND

This request allows a protocol to send data destined for another protocol module, not a user. The details of how data is marked "addressed to protocol" instead of "addressed to user" are left to the protocol modules. No protocols currently use this facility.

8. Protocol/protocol interface

The interface between protocol modules is through the *pr_usrreq*, *pr_input*, *pr_output*, *pr_ctlinput*, and *pr_ctloutput* routines. The calling conventions for all but the *pr_usrreq* routine are expected to be specific to the protocol modules and are not guaranteed to be consistent across protocol families. We will examine the conventions used for some of the Internet protocols in this section as an example.

8.1. pr_output

The Internet protocol UDP uses the convention,

```
error = udp_output(inp, m);
int error; struct inpcb *inp; struct mbuf *m;
```

where the *inp*, "internet protocol control block", passed between modules conveys per connection state information, and the mbuf chain contains the data to be sent. UDP performs consistency checks, appends its header, calculates a checksum, etc. before passing the packet on to the IP module:

```
error = ip_output(m, opt, ro, allowbroadcast);
int error; struct mbuf *m, *opt; struct route *ro; int allowbroadcast;
```

The call to IP's output routine is more complicated than that for UDP, as befits the additional work the IP module must do. The *m* parameter is the data to be sent, and the *opt* parameter is an optional list of IP options which should be placed in the IP packet header. The *ro* parameter is is used in making routing decisions (and passing them back to the caller). The final parameter, *allowbroadcast* is a flag indicating if the user is allowed to transmit a broadcast packet. This may be inconsequential if the underlying hardware does not support the notion of broadcasting.

All output routines return 0 on success and a UNIX error number if a failure occured which could be immediately detected (no buffer space available, no route to destination, etc.).

8.2. pr_input

Both UDP and TCP use the following calling convention,

```
(void) (*protosw[].pr_input)(m);
struct mbuf *m;
```

Each mbuf list passed is a single packet to be processed by the protocol module.

The IP input routine is a VAX software interrupt level routine, and so is not called with any parameters. It instead communicates with network interfaces through a queue, *ipintrq*, which is identical in structure to the queues used by the network interfaces for storing packets awaiting transmission.

8.3. pr_ctlinput

This routine is used to convey "control" information to a protocol module (i.e. information which might be passed to the user, but is not data). This routine, and the *pr_ctloutput* routine, have not been extensively developed, and thus suffer from a "clumsiness" that can only be improved as more demands are placed on it.

The common calling convention for this routine is,

```
(void) (*protosw[].pr_ctlinput)(req, info);
int req; caddr_t info;
```

The *req* parameter is one of the following,

```
#define  PRC_IFDOWN              0    /* interface transition */
#define  PRC_ROUTEDEAD           1    /* select new route if possible */
#define  PRC_QUENCH              4    /* some said to slow down */
#define  PRC_HOSTDEAD            6    /* normally from IMP */
#define  PRC_HOSTUNREACH         7    /* ditto */
#define  PRC_UNREACH_NET         8    /* no route to network */
#define  PRC_UNREACH_HOST        9    /* no route to host */
#define  PRC_UNREACH_PROTOCOL    10   /* dst says bad protocol */
#define  PRC_UNREACH_PORT        11   /* bad port # */
#define  PRC_MSGSIZE             12   /* message size forced drop */
#define  PRC_REDIRECT_NET        13   /* net routing redirect */
#define  PRC_REDIRECT_HOST       14   /* host routing redirect */
#define  PRC_TIMXCEED_INTRANS    17   /* packet lifetime expired in transit */
#define  PRC_TIMXCEED_REASS      18   /* lifetime expired on reass q */
#define  PRC_PARAMPROB           19   /* header incorrect */
```

while the *info* parameter is a "catchall" value which is request dependent. Many of the requests have obviously been derived from ICMP (the Internet Control Message Protocol), and from error messages defined in the 1822 host/IMP convention [BBN78]. Mapping tables exist to convert control requests to UNIX error codes which are delivered to a user.

8.4. pr_ctloutput

This routine is not currently used by any protocol modules.

9. Protocol/network-interface interface

The lowest layer in the set of protocols which comprise a protocol family must interface itself to one or more network interfaces in order to transmit and receive packets. It is assumed that any routing decisions have been made before handing a packet to a network interface, in fact this is absolutely necessary in order to locate any interface at all (unless, of course, one uses a single "hardwired" interface). There are two cases to be concerned with, transmission of a packet, and receipt of a packet; each will be considered separately.

9.1. Packet transmission

Assuming a protocol has a handle on an interface, *ifp*, a (struct ifnet *), it transmits a fully formatted packet with the following call,

```
error = (*ifp->if_output)(ifp, m, dst)
int error; struct ifnet *ifp; struct mbuf *m; struct sockaddr *dst;
```

The output routine for the network interface transmits the packet *m* to the *dst* address, or returns an error indication (a UNIX error number). In reality transmission may not be immediate, or successful; normally the output routine simply queues the packet on its send queue and primes an

interrupt driven routine to actually transmit the packet. For unreliable mediums, such as the Ethernet, "successful" transmission simply means the packet has been placed on the cable without a collision. On the other hand, an 1822 interface guarantees proper delivery or an error indication for each message transmitted. The model employed in the networking system attaches no promises of delivery to the packets handed to a network interface, and thus corresponds more closely to the Ethernet. Errors returned by the output routine are normally trivial in nature (no buffer space, address format not handled, etc.).

9.2. Packet reception

Each protocol family must have one or more "lowest level" protocols. These protocols deal with internetwork addressing and are responsible for the delivery of incoming packets to the proper protocol processing modules. In the PUP model [Boggs78] these protocols are termed Level 1 protocols, in the ISO model, network layer protocols. In our system each such protocol module has an input packet queue assigned to it. Incoming packets received by a network interface are queued up for the protocol module and a VAX software interrupt is posted to initiate processing.

Three macros are available for queueing and dequeueing packets,

IF_ENQUEUE(ifq, m)
This places the packet *m* at the tail of the queue *ifq*.

IF_DEQUEUE(ifq, m)
This places a pointer to the packet at the head of queue *ifq* in *m*. A zero value will be returned in *m* if the queue is empty.

IF_PREPEND(ifq, m)
This places the packet *m* at the head of the queue *ifq*.

Each queue has a maximum length associated with it as a simple form of congestion control. The macro IF_QFULL(ifq) returns 1 if the queue is filled, in which case the macro IF_DROP(ifq) should be used to bump a count of the number of packets dropped and the offending packet dropped. For example, the following code fragment is commonly found in a network interface's input routine,

```
if (IF_QFULL(inq)) {
        IF_DROP(inq);
        m_freem(m);
} else
        IF_ENQUEUE(inq, m);
```

10. Gateways and routing issues

The system has been designed with the expectation that it will be used in an internetwork environment. The "canonical" environment was envisioned to be a collection of local area networks connected at one or more points through hosts with multiple network interfaces (one on each local area network) and possibly a connection to a long haul network (for example, the ARPANET). In such an environment, issues of gatewaying and packet routing become very important. Certain of these issues, such as congestion control, have been handled in a simplistic manner or specifically not addressed. Instead, where possible, the network system attempts to provide simple mechanisms upon which more involved policies may be implemented. As some of these problems become better understood, the solutions developed will be incorporated into the system.

This section will describe the facilities provided for packet routing. The simplistic mechanisms provided for congestion control are described in chapter 12.

10.1. Routing tables

The network system maintains a set of routing tables for selecting a network interface to use in delivering a packet to its destination. These tables are of the form:

```
struct rtentry {
        u_long      rt_hash;        /* hash key for lookups */
        struct      sockaddr rt_dst; /* destination net or host */
        struct      sockaddr rt_gateway; /* forwarding agent */
        short rt_flags;         /* see below */
        short rt_refcnt;        /* no. of references to structure */
        u_long      rt_use;             /* packets sent using route */
        struct      ifnet *rt_ifp;      /* interface to give packet to */
};
```

The routing information is organized in two separate tables, one for routes to a host and one for routes to a network. The distinction between hosts and networks is necessary so that a single mechanism may be used for both broadcast and multi-drop type networks, and also for networks built from point-to-point links (e.g DECnet [DEC80]).

Each table is organized as a hashed set of linked lists. Two 32-bit hash values are calculated by routines defined for each address family; one based on the destination being a host, and one assuming the target is the network portion of the address. Each hash value is used to locate a hash chain to search (by taking the value modulo the hash table size) and the entire 32-bit value is then used as a key in scanning the list of routes. Lookups are applied first to the routing table for hosts, then to the routing table for networks. If both lookups fail, a final lookup is made for a "wildcard" route (by convention, network 0). By doing this, routes to a specific host on a network may be present as well as routes to the network. This also allows a "fall back" network route to be defined to an "smart" gateway which may then perform more intelligent routing.

Each routing table entry contains a destination (who's at the other end of the route), a gateway to send the packet to, and various flags which indicate the route's status and type (host or network). A count of the number of packets sent using the route is kept for use in deciding between multiple routes to the same destination (see below), and a count of "held references" to the dynamically allocated structure is maintained to insure memory reclamation occurs only when the route is not in use. Finally a pointer to the a network interface is kept; packets sent using the route should be handed to this interface.

Routes are typed in two ways: either as host or network, and as "direct" or "indirect". The host/network distinction determines how to compare the *rt_dst* field during lookup. If the route is to a network, only a packet's destination network is compared to the *rt_dst* entry stored in the table. If the route is to a host, the addresses must match bit for bit.

The distinction between "direct" and "indirect" routes indicates whether the destination is directly connected to the source. This is needed when performing local network encapsulation. If a packet is destined for a peer at a host or network which is not directly connected to the source, the internetwork packet header will indicate the address of the eventual destination, while the local network header will indicate the address of the intervening gateway. Should the destination be directly connected, these addresses are likely to be identical, or a mapping between the two exists. The RTF_GATEWAY flag indicates the route is to an "indirect" gateway agent and the local network header should be filled in from the *rt_gateway* field instead of *rt_dst*, or from the internetwork destination address.

It is assumed multiple routes to the same destination will not be present unless they are deemed *equal* in cost (the current routing policy process never installs multiple routes to the same destination). However, should multiple routes to the same destination exist, a request for a route will return the "least used" route based on the total number of packets sent along this route. This can result in a "ping-pong" effect (alternate packets taking alternate routes), unless protocols "hold onto" routes until they no longer find them useful; either because the destination has

changed, or because the route is lossy.

Routing redirect control messages are used to dynamically modify existing routing table entries as well as dynamically create new routing table entries. On hosts where exhaustive routing information is too expensive to maintain (e.g. work stations), the combination of wildcard routing entries and routing redirect messages can be used to provide a simple routing management scheme without the use of a higher level policy process. Statistics are kept by the routing table routines on the use of routing redirect messages and their affect on the routing tables. These statistics may be viewed using

Status information other than routing redirect control messages may be used in the future, but at present they are ignored. Likewise, more intelligent "metrics" may be used to describe routes in the future, possibly based on bandwidth and monetary costs.

10.2. Routing table interface

A protocol accesses the routing tables through three routines, one to allocate a route, one to free a route, and one to process a routing redirect control message. The routine *rtalloc* performs route allocation; it is called with a pointer to the following structure,

```
struct route {
        struct      rtentry *ro_rt;
        struct      sockaddr ro_dst;
};
```

The route returned is assumed "held" by the caller until disposed of with an *rtfree* call. Protocols which implement virtual circuits, such as TCP, hold onto routes for the duration of the circuit's lifetime, while connection-less protocols, such as UDP, currently allocate and free routes on each transmission.

The routine *rtredirect* is called to process a routing redirect control message. It is called with a destination address and the new gateway to that destination. If a non-wildcard route exists to the destination, the gateway entry in the route is modified to point at the new gateway supplied. Otherwise, a new routing table entry is inserted reflecting the information supplied. Routes to interfaces and routes to gateways which are not directly accesible from the host are ignored.

10.3. User level routing policies

Routing policies implemented in user processes manipulate the kernel routing tables through two *ioctl* calls. The commands SIOCADDRT and SIOCDELRT add and delete routing entries, respectively; the tables are read through the /dev/kmem device. The decision to place policy decisions in a user process implies routing table updates may lag a bit behind the identification of new routes, or the failure of existing routes, but this period of instability is normally very small with proper implementation of the routing process. Advisory information, such as ICMP error messages and IMP diagnostic messages, may be read from raw sockets (described in the next section).

One routing policy process has already been implemented. The system standard "routing daemon" uses a variant of the Xerox NS Routing Information Protocol [Xerox82] to maintain up to date routing tables in our local environment. Interaction with other existing routing protocols, such as the Internet GGP (Gateway-Gateway Protocol), may be accomplished using a similar process.

11. Raw sockets

A raw socket is a mechanism which allows users direct access to a lower level protocol. Raw sockets are intended for knowledgeable processes which wish to take advantage of some protocol

feature not directly accessible through the normal interface, or for the development of new protocols built atop existing lower level protocols. For example, a new version of TCP might be developed at the user level by utilizing a raw IP socket for delivery of packets. The raw IP socket interface attempts to provide an identical interface to the one a protocol would have if it were resident in the kernel.

The raw socket support is built around a generic raw socket interface, and (possibly) augmented by protocol-specific processing routines. This section will describe the core of the raw socket interface.

11.1. Control blocks

Every raw socket has a protocol control block of the following form,

```
struct rawcb {
        struct      rawcb *rcb_next;        /* doubly linked list */
        struct      rawcb *rcb_prev;
        struct      socket *rcb_socket;     /* back pointer to socket */
        struct      sockaddr rcb_faddr;     /* destination address */
        struct      sockaddr rcb_laddr;     /* socket's address */
        caddr_t     rcb_pcb;                /* protocol specific stuff */
        short       rcb_flags;
};
```

All the control blocks are kept on a doubly linked list for performing lookups during packet dispatch. Associations may be recorded in the control block and used by the output routine in preparing packets for transmission. The addresses are also used to filter packets on input; this will be described in more detail shortly. If any protocol specific information is required, it may be attached to the control block using the *rcb_pcb* field.

A raw socket interface is datagram oriented. That is, each send or receive on the socket requires a destination address. This address may be supplied by the user or stored in the control block and automatically installed in the outgoing packet by the output routine. Since it is not possible to determine whether an address is present or not in the control block, two flags, RAW_LADDR and RAW_FADDR, indicate if a local and foreign address are present. Another flag, RAW_DONTROUTE, indicates if routing should be performed on outgoing packets. If it is, a route is expected to be allocated for each "new" destination address. That is, the first time a packet is transmitted a route is determined, and thereafter each time the destination address stored in *rcb_route* differs from *rcb_faddr*, or *rcb_route.ro_rt* is zero, the old route is discarded and a new one allocated.

11.2. Input processing

Input packets are "assigned" to raw sockets based on a simple pattern matching scheme. Each network interface or protocol gives packets to the raw input routine with the call:

```
raw_input(m, proto, src, dst)
struct mbuf *m; struct sockproto *proto, struct sockaddr *src, *dst;
```

The data packet then has a generic header prepended to it of the form

```
struct raw_header {
        struct      sockproto raw_proto;
        struct      sockaddr raw_dst;
        struct      sockaddr raw_src;
};
```

and it is placed in a packet queue for the "raw input protocol" module. Packets taken from this queue are copied into any raw sockets that match the header according to the following rules,

1) The protocol family of the socket and header agree.
2) If the protocol number in the socket is non-zero, then it agrees with that found in the packet header
3) If a local address is defined for the socket, the address format of the local address is the same as the destination address's and the two addresses agree bit for bit.
4) The rules of 3) are applied to the socket's foreign address and the packet's source address.

A basic assumption is that addresses present in the control block and packet header (as constructed by the network interface and any raw input protocol module) are in a canonical form which may be "block compared".

11.3. Output processing

On output the raw *pr_usrreq* routine passes the packet and raw control block to the raw protocol output routine for any processing required before it is delivered to the appropriate network interface. The output routine is normally the only code required to implement a raw socket interface.

12. Buffering and congestion control

One of the major factors in the performance of a protocol is the buffering policy used. Lack of a proper buffering policy can force packets to be dropped, cause falsified windowing information to be emitted by protocols, fragment host memory, degrade the overall host performance, etc. Due to problems such as these, most systems allocate a fixed pool of memory to the networking system and impose a policy optimised for "normal" network operation.

The networking system developed for UNIX is little different in this respect. At boot time a fixed amount of memory is allocated by the networking system. At later times more system memory may be requested as the need arises, but at no time is memory ever returned to the system. It is possible to garbage collect memory from the network, but difficult. In order to perform this garbage collection properly, some portion of the network will have to be "turned off" as data structures are updated. The interval over which this occurs must kept small compared to the average inter-packet arrival time, or too much traffic may be lost, impacting other hosts on the network, as well as increasing load on the interconnecting mediums. In our environment we have not experienced a need for such compaction, and thus have left the problem unresolved.

The mbuf structure was introduced in chapter 5. In this section a brief description will be given of the allocation mechanisms, and policies used by the protocols in performing connection level buffering.

12.1. Memory management

The basic memory allocation routines place no restrictions on the amount of space which may be allocated. Any request made is filled until the system memory allocator starts refusing to allocate additional memory. When the current quota of memory is insufficient to satisfy an mbuf allocation request, the allocator requests enough new pages from the system to satisfy the current request only. All memory owned by the network is described by a private page table used in remapping pages to be logically contiguous as the need arises. In addition, an array of reference counts parallels the page table and is used when multiple copies of a page are present.

Mbufs are 128 byte structures, 8 fitting in a 1Kbyte page of memory. When data is placed in mbufs, if possible, it is copied or remapped into logically contiguous pages of memory from the network page pool. Data smaller than the size of a page is copied into one or more 112 byte mbuf data areas.

12.2. Protocol buffering policies

Protocols reserve fixed amounts of buffering for send and receive queues at socket creation time. These amounts define the high and low water marks used by the socket routines in deciding when to block and unblock a process. The reservation of space does not currently result in any action by the memory management routines, though it is clear if one imposed an upper bound on the total amount of physical memory allocated to the network, reserving memory would become important.

Protocols which provide connection level flow control do this based on the amount of space in the associated socket queues. That is, send windows are calculated based on the amount of free space in the socket's receive queue, while receive windows are adjusted based on the amount of data awaiting transmission in the send queue. Care has been taken to avoid the "silly window syndrome" described in [Clark82] at both the sending and receiving ends.

12.3. Queue limiting

Incoming packets from the network are always received unless memory allocation fails. However, each Level 1 protocol input queue has an upper bound on the queue's length, and any packets exceeding that bound are discarded. It is possible for a host to be overwhelmed by excessive network traffic (for instance a host acting as a gateway from a high bandwidth network to a low bandwidth network). As a "defensive" mechanism the queue limits may be adjusted to throttle network traffic load on a host. Consider a host willing to devote some percentage of its machine to handling network traffic. If the cost of handling an incoming packet can be calculated so that an acceptable "packet handling rate" can be determined, then input queue lengths may be dynamically adjusted based on a host's network load and the number of packets awaiting processing. Obviously, discarding packets is not a satisfactory solution to a problem such as this (simply dropping packets is likely to increase the load on a network); the queue lengths were incorporated mainly as a safeguard mechanism.

12.4. Packet forwarding

When packets can not be forwarded because of memory limitations, the system generates a "source quench" message. In addition, any other problems encountered during packet forwarding are also reflected back to the sender in the form of ICMP packets. This helps hosts avoid unneeded retransmissions.

Broadcast packets are never forwarded due to possible dire consequences. In an early stage of network development, broadcast packets were forwarded and a "routing loop" resulted in network saturation and every host on the network crashing.

13. Out of band data

Out of band data is a facility peculiar to the stream socket abstraction defined. Little agreement appears to exist as to what its semantics should be. TCP defines the notion of "urgent data" as in-line, while the NBS protocols [Burruss81] and numerous others provide a fully independent logical transmission channel along which out of band data is to be sent. In addition, the amount of the data which may be sent as an out of band message varies from protocol to protocol; everything from 1 bit to 16 bytes or more.

A stream socket's notion of out of band data has been defined as the lowest reasonable common denominator (at least reasonable in our minds); clearly this is subject to debate. Out of band data is expected to be transmitted out of the normal sequencing and flow control constraints of the data stream. A minimum of 1 byte of out of band data and one outstanding out of band message are expected to be supported by the protocol supporting a stream socket. It is a protocols prerogative to support larger sized messages, or more than one outstanding out of band message at a time.

Out of band data is maintained by the protocol and usually not stored in the socket's send queue. The PRU_SENDOOB and PRU_RCVOOB requests to the *pr_usrreq* routine are used in sending and receiving data.

14. Trailer protocols

Core to core copies can be expensive. Consequently, a great deal of effort was spent in minimizing such operations. The VAX architecture provides virtual memory hardware organized in page units. To cut down on copy operations, data is kept in page sized units on page-aligned boundaries whenever possible. This allows data to be moved in memory simply by remapping the page instead of copying. The mbuf and network interface routines perform page table manipulations where needed, hiding the complexities of the VAX virtual memory hardware from higher level code.

Data enters the system in two ways: from the user, or from the network (hardware interface). When data is copied from the user's address space into the system it is deposited in pages (if sufficient data is present to fill an entire page). This encourages the user to transmit information in messages which are a multiple of the system page size.

Unfortunately, performing a similar operation when taking data from the network is very difficult. Consider the format of an incoming packet. A packet usually contains a local network header followed by one or more headers used by the high level protocols. Finally, the data, if any, follows these headers. Since the header information may be variable length, DMA'ing the eventual data for the user into a page aligned area of memory is impossible without a priori knowledge of the format (e.g. supporting only a single protocol header format).

To allow variable length header information to be present and still ensure page alignment of data, a special local network encapsulation may be used. This encapsulation, termed a *trailer protocol*, places the variable length header information after the data. A fixed size local network header is then prepended to the resultant packet. The local network header contains the size of the data portion, and a new *trailer protocol header*, inserted before the variable length information, contains the size of the variable length header information. The following trailer protocol header is used to store information regarding the variable length protocol header:

```
struct {
        short protocol;   /* original protocol no. */
        short length;     /* length of trailer */
};
```

The processing of the trailer protocol is very simple. On output, the local network header indicates a trailer encapsulation is being used. The protocol identifier also includes an indication of the number of data pages present (before the trailer protocol header). The trailer protocol header is initialized to contain the actual protocol and variable length header size, and appended to the data along with the variable length header information.

On input, the interface routines identify the trailer encapsulation by the protocol type stored in the local network header, then calculate the number of pages of data to find the beginning of the trailer. The trailing information is copied into a separate mbuf and linked to the front of the resultant packet.

Clearly, trailer protocols require cooperation between source and destination. In addition, they are normally cost effective only when sizable packets are used. The current scheme works because the local network encapsulation header is a fixed size, allowing DMA operations to be performed at a known offset from the first data page being received. Should the local network header be variable length this scheme fails.

Statistics collected indicate as much as 200Kb/s can be gained by using a trailer protocol with 1Kbyte packets. The average size of the variable length header was 40 bytes (the size of a

minimal TCP/IP packet header). If hardware supports larger sized packets, even greater gains may be realized.

References – Unix System Enhancements

[Accetta80] Accetta, M., Robertson, G., Satyanarayanan, M., and Thompson, M. "The Design of a Network-Based Central File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-134

[Almes78] Almes, G., and Robertson, G. "An Extensible File System for Hydra" Proceedings of the Third International Conference on Software Engineering, IEEE, May 1978.

[Bass81] Bass, J. "Implementation Description for File Locking", Onyx Systems Inc, 73 E. Trimble Rd, San Jose, CA 95131 Jan 1981.

[Boggs79] Boggs, D. R., J. F. Shoch, E. A. Taft, and R. M. Metcalfe; *PUP: An Internetwork Architecture*. Report CSL-79-10. XEROX Palo Alto Research Center, July 1979.

[BBN78] Bolt Beranek and Newman; *Specification for the Interconnection of Host and IMP*. BBN Technical Report 1822. May 1978.

[Cerf78] Cerf, V. G.; The Catenet Model for Internetworking. Internet Working Group, IEN 48. July 1978.

[Clark82] Clark, D. D.; Window and Acknowledgement Strategy in TCP. Internet Working Group, IEN Draft Clark-2. March 1982.

[DEC80] Digital Equipment Corporation; *DECnet DIGITAL Network Architecture – General Description*. Order No. AA-K179A-TK. October 1980.

[Dion80] Dion, J. "The Cambridge File Server", Operating Systems Review, 14, 4. Oct 1980. pp 26-35

[Eswaran74] Eswaran, K. "Placement of records in a file and file allocation in a computer network", Proceedings IFIPS, 1974. pp 304-307

[Feiertag71] Feiertag, R. J. and Organick, E. I., "The Multics Input-Output System", Proceedings of the Third Symposium on Operating Systems Principles, ACM, Oct 1971. pp 35-41

[Gurwitz81] Gurwitz, R. F.; VAX-UNIX Networking Support Project – Implementation Description. Internetwork Working Group, IEN 168. January 1981.

[Holler73] Holler, J. "Files in Computer Networks", First European Workshop on Computer Networks, April 1973. pp 381-396

[ISO81] International Organisation for Standardization. *ISO Open Systems Interconnection – Basic Reference Model*. ISO/TC 97/SC 16 N 719. August 1981.

[Kowalski78] Kowalski, T. "FSCK - The UNIX System Check Program", Bell Laboratory, Murray Hill, NJ 07974. March 1978

[Kridle83] Kridle, R., and McKusick, M., "Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX", Computer Systems Research Group, Dept of EECS, Berkeley, CA 94720, Technical Report #8.

[Luniewski77] Luniewski, A. "File Allocation in a Distributed System", MIT Laboratory for Computer Science, Dec 1977.

[Maruyama76] Maruyama, K., and Smith, S. "Optimal reorganization of Distributed Space Disk Files", Communications of the ACM, 19, 11. Nov 1976. pp 634-642

[Nevalainen77] Nevalainen, O., Vesterinen, M. "Determining Blocking Factors for Sequential Files by Heuristic Methods", The Computer Journal, 20, 3. Aug 1977. pp 245-247

[Peterson83] Peterson, G. "Concurrent Reading While Writing", ACM Transactions on Programming Languages and Systems, ACM, 5, 1. Jan 1983. pp 46-55

[Porcar82] Porcar, J. "File Migration in Distributed Computer Systems", Ph.D. Thesis, Lawrence Berkeley Laboratory Tech Report #LBL-14763.

[Postel79] Postel, J., ed. *DOD Standard User Datagram Protocol.* Internet Working Group, IEN 88. May 1979.

[Postel80a] Postel, J., ed. *DOD Standard Internet Protocol.* Internet Working Group, IEN 128. January 1980.

[Postel80b] Postel, J., ed. *DOD Standard Transmission Control Protocol.* Internet Working Group, IEN 129. January 1980.

[Powell79] Powell, M. "The DEMOS File System", Proceedings of the Sixth Symposium on Operating Systems Principles, ACM, Nov 1977. pp 33-42

[Ritchie74] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", CACM 17, 7. July 1974. pp 365-375

[Smith81a] Smith, A. "Input/Output Optimization and Disk Architectures: A Survey", Performance and Evaluation 1. Jan 1981. pp 104-117

[Smith81b] Smith, A. "Bibliography on File and I/O System Optimization and Related Topics", Operating Systems Review, 15, 4. Oct 1981. pp 39-54

[Sturgis80] Sturgis, H., Mitchell, J., and Israel, J. "Issues in the Design and Use of a Distributed File System", Operating Systems Review, 14, 3. pp 55-79

[Symbolics81a] "Symbolics File System", Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Aug 1981.

[Symbolics81b] "Chaosnet FILE Protocol". Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Sept 1981.

[Thompson79] Thompson, K. "UNIX Implementation", Section 31, Volume 2B, UNIX Programmers Manual, Bell Laboratory, Murray Hill, NJ 07974. Jan 1979

[Thompson80] Thompson, M. "Spice File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-???

[Trivedi80] Trivedi, K. "Optimal Selection of CPU Speed, Device Capabilities, and File Assignments", Journal of the ACM, 27, 3. July 1980. pp 457-473

[White80] White, R. M. "Disk Storage Technology", Scientific American, 243(2), August 1980.

[Xerox81] Xerox Corporation. *Internet Transport Protocols.* Xerox System Integration Standard 028112. December 1981.

[Zimmermann80] Zimmermann, H. OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. IEEE Transactions on Communications. Com-28(4); 425-432. April 1980.

References – Human/Machine Interaction and Expert Database Systems

[1] Brian A. Barsky, "The Beta Spline: A Local Representation Based on Shape Parameters and Fundamental Geometric Measures," Ph.D. Dissertation, University of Utah, Salt Lake City, Utah (December, 1981).

[2] Edwin E. Catmull and Raphael J. Rom, "A Class of Local Interpolating Splines," *Computer Aided Geometric Design*, Barnhill, Robert E. and Riesenfeld, Richard F. (ed.), Academic Press, New York (1974). pp. 317-326.

[3] Tony D. DeRose and Brian A. Barsky, "Geometric Continuity and Shape Parameters for Catmull-Rom Splines (Extended Abstract)", *Proceedings of Graphics Interface '84*, Ottawa, (27 May - 1 June 1984), pp. 57-64.

[4] Robert Wilensky, "Knowledge Representation - A Critique and a Proposal", *Proceedings of the First Annual Workshop on Conceptual Information Processing*, Atlanta, Georgia, (1984).

[5] Robert Wilensky, "A Knowledge Representation Language", *Proceedings of the 6th. National Conference of the Cognitive Science Society*, (April 1984).

[5] Paul Jacobs, "PHRED: A Generator for Natural Language Interfaces", Technical Report, Computer Science Division, EECS, University of California, Berkeley, Report No. UCB/CSD 85/198, (January 1985).

[6] Richard Alterman, "Plexus: Networks for Adaptive Planning". *Proceedings of the Second Workshop on Theoretical Issues in Conceptual Information Processing*. New Haven, CT. (May, 1985).

[6] Richard Alterman, "Text Concept Coherence", To appear in: *Advances in Natural Language Processing*, David Waltz (ed.). Lawrence Erlbaum Associates.

[6] Richard Alterman, "Adaptive Planning: Refitting Old Planning Experiences to New Situations", *The Seventh Annual Conference of the Cognitive Science Society*, (1985).

[7] Zadeh, L.A., "A Computational Approach to Fuzzy Quantifiers in Natural Languages", *Computers and Mathematics 9* (1983) pp. 149-184.

[8] Zadeh, L.A., "The Role of Fuzzy Logic in the Management of Uncertainty in Expert Systems", *Fuzzy Sets and Systems 11*, (1983), pp. 199-227.

[9] Linton, M., "Queries and Views of Programs Using a Relational Database System", Report No. UCB/CSD 83/164, Computer Science Division, University of California, Berkeley, California, (December 1983).

[10] Habermann, A. N., Ellison, E., Medina-Mora, R., Feiler, P., Notkin, D., Kaiser, G. E., Garlan, D. B., and Popovich, S., "The Second Compendium of Gandalf Documentation", CMU Department of Computer Science, (May 1982).

[11] Stonebraker, M., Wong, E., and Kreps, P., "The Design and Implementation of INGRES", *ACM Transactions on Database Systems 1*, 3, (September 1976).

[12] Carl Meyer, "A Browser for Directed Graphs", M.S. Project Report, Computer Science Division, EECS, University of California, Berkeley, (December 1983).

[13] Robert R. Henry, "Graham-Glanville Code Generators", PhD Dissertation, Computer Science Division, EECS, University of California, Berkeley, Report No. UCB/CSD 84/184, (May 1984).

[14] R. Giegerich, "A Formal Framework for the Derivation of Machine-Specific Optimizers", *ACM Transactions on Programming Languages and Systems 5*, 3, (July 1983), pp. 478-498.

[15] Peter B. Kessler, "Automated Discovery of Machine Specific Code Improvements", PhD Dissertation, Computer Science Division, EECS, University of California, Berkeley, Report No. UCB/CSD 84/213, (December, 1984).

[16] J. W. Davidson and C. W. Fraser, "The Design and Application of a Retargetable Peephole Optimizer", *ACM Transactions on Programming Languages and Systems 2*, 2, (April 1980), pp.

191-202.

[17] J. W. Davidson and C. W. Fraser, "Automatic Generation of Peephole Optimizations," *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction*, Montreal, Quebec, SIGPLAN Notices 19, 6, (June 1984), pp. 111-116.

[18] Robert R. Kessler, "Peep - An Architectural Description Driven Peephole Optimizer", *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction*, Montreal, Quebec, SIGPLAN Notices 19, 6, (June 1984), pp. 106-110.

[19] Benjamin G. Zorn, "Experiences with Ada Code Generation", Technical Report, Computer Science Division, EECS, University of California, Berkeley, Report No. UCB/CSD 85/249, (June 1985).

[20] M. P. Murphy, "DIDI - Diana Implentation Designs and Decisions", Master's Thesis, Computer Science Division, EECS, UCB, Berkeley, CA, (1984), pp 16-17.

[21] Dain Samples, David Ungar, Paul Hilfinger, "SOAR: Smalltalk without Bytecodes", *Conference on Object-Oriented Programming Systems, Languages and Applications Proceedings*, SIGPLAN Notices 21, 11, (Nov. 1986), pp. 107-118.

[22] G. Kahn, B. Lang, B. Melese, and E. Morcos. "Metal: a formalism for specifying formalisms." *Science of Programming* 3 (1983) pp. 151-188.

[23] C. Zaniolo, "The Database Language GEM", *Proc. 1983 ACM-SIGMOD Conference on Management of Data*, San Jose, CA, (May 1983).

[24] Michael Stonebraker et. al., "The Design and Implementation of INGRES," *ACM Transactions on Database Systems 2*, 3, (September 1976).

[25] Michael Stonebraker et. al., "Extending a Database System With Procedures", (to appear in ACM Transactions on Database Systems).

[26] R. Kooi, D. Frankfurth, "Query Optimization in INGRES," *Database Engineering*, (September 1982).

[27] P. Selinger, "Access Path Selection in a Relation Database System," *Proc. 1979 ACM SIGMOD Conference on Management of Data*, Boston, Mass., (June 1979).

[28] E. Wong, "Decomposition: A Strategy for Query Processing," *ACM Transactions on Database Systems*, (September 1976).